HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTERARBEIT

Herr
Christian Donath

Programmierung eines Graphischen User Interfaces (GUI) in Matlab zur Ansteuerung einer schnell-laufenden Permanenterregten Synchronmaschine (PESM)

2019

Fakultät Ingenieurwissenschaften

MASTERARBEIT

Programmierung eines Graphischen User Interfaces (GUI) in Matlab zur Ansteuerung einer schnell-laufenden Permanenterregten Synchronmaschine (PESM)

Autor:
Herr Christian Donath

Studiengang:
Elektro- und Informationstechnik

Seminargruppe:
EI17w1-M

Erstprüfer:
Prof. Dr.-Ing. Lutz Rauchfuß

Zweitprüfer:
M. Sc. Jan Roloff

Einreichung:
Mittweida, 31.08.2019

Faculty Engineering Sciences

MASTER THESIS

Programming a Graphical User Interface (GUI) with Matlab to controll a high-speed Permanently Excited Synchronous Machine (PESM)

author:
Herr Christian Donath

course of studies:
Electro and Information Technology

seminar group:
EI17w1-M

first examiner:
Prof. Dr.-Ing. Lutz Rauchfuß

second examiner:
M. Sc. Jan Roloff

submission:
Mittweida, 31.08.2019

Bibliografische Beschreibung:

Donath, Christian:
Programmierung eines Graphischen User Interfaces (GUI) in Matlab zur Ansteuerung einer schnelllaufenden Permanenterregten Synchronmaschine (PESM). - 2019. - 51 Seiten
Mittweida, Hochschule Mittweida, Fakultät Ingenieurwissenschaften, Masterarbeit, 2019

Referat:

Diese Arbeit beschäftigt sich mit dem grundlegenden Aufbau eines Motorprüfstandes. Im ersten Teil werden die verwendeten Geräte, Baugruppen und Schaltungen dargestellt. Der zweite Teil zeigt die Software der GUI, die eine automatische Aufnahme der Drehzahl-Drehmomenten-Kennlinie einer elektrischen Maschine durchführen kann.

Fakultät, Ingenieurwissenschaften (INW)
Professur Antriebs- und Regelungstechnik (ART)
Prof. Dr.-Ing. Lutz Rauchfuß

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Masterthesis für Christian Donath

Mittweida, 17.06.2019

Thema: Programmierung eines Graphischen User Interfaces (GUI) in Matlab zur Ansteuerung einer schnelllaufenden Permanenterregten Synchronmaschine (PESM)

Für die Ausbildung der Studenten ist es erforderlich ständig neue Versuchsstände zu entwickeln, die sich am aktuellen Stand der Technik orientieren. Die PESM wird zu einem großen Teil in der Elektromobilität, aber auch in Hybridfahrzeugen genutzt. Um die Maschine und deren Inverter transparent ansteuern zu können, bietet sich ein GUI in Matlab an. Da der Inverter der Firma Bosch nur über den CAN-Bus steuerbar ist, muss eine geeignete Schnittstelle zwischen PC und Inverter gefunden werden, darüber Parameter und Daten austauschen zu können.

Arbeitspakete:

1. Auswahl der geeigneten Schnittstelle zwischen Inverter und PC
   - CAN-Bus konforme Hard- und Software beschaffen
   - Initialisierung der BUS-Teilnehmer
   - Inbetriebnahme CAN-Bus
2. Entwicklung des GUI in Matlab
   - Wahlmöglichkeit von Momenten- und Drehzahlregelung
   - Anzeige der Soll- und Istwerte von Drehzahl und Moment
   - Auswahl weiterer Maschinendaten die vom CAN-Bus gelesen werden können, die den Betriebszustand der Maschine transparent machen
   - Vorgabe der Sollwerte als Zahlenwert, Funktion (Sinus + Offset) und als Schieberegler
   - Programmierung geeigneter Rampen- und Filterfunktionen (FIR, biharmonisch $\sin^2 x$)
   - Option: Entwicklung des Drehzahlreglers (PID) für die PESM in Matlab/CANoe
   - Darstellung der aufgenommenen n/M-Daten in einem Maschinen-Kennfeld
3. Inbetriebnahme des Inverters der PESM über das GUI
   - Stückweise Inbetriebnahme unter Berücksichtigung der Betriebsart (n, M-Regelung) Bremsvorrichtung, Wasserkühlung der PESM+Inverter (Sicherheit)
   - Plausibilisierung der Soll- und Istwerte vom GUI mit Hilfe geeigneter Messgeräte

Hochschulbetreuer:

Prof. Dr.-Ing. Lutz Rauchfuß

Inhalt

Inhalt I

Abbildungsverzeichnis III

Tabellenverzeichnis V

Abkürzungsverzeichnis VI

1 Einleitung 1

1.1 Hintergrund und Aufgabenstellung 1

1.2 Aufbau 1

2 Hardware 2

2.1 Übersicht 2

2.2 Maschinensatz 3

2.3 DC-Netzgerät 5

2.4 Relais 6

2.5 CAN-Interface 8

2.6 Schaltung 9

2.6.1 Erdung 11

2.6.2 Kühlung 12

3 Software 13

3.1 CANoe 13

3.1.1 Hardware-Verknüpfung 15

3.1.2 Werkzeuge 16

3.1.2.1 Panel Designer 16

3.1.2.2 CAPL Browser 17

3.1.3 Systemvariablen 17

3.2 GUI 18

3.3 CAPL Skripte 19

3.3.1 Systemstart 19

3.3.2 Nachrichten 20

3.3.3 Aktivierung / Betriebsart 21

3.3.4 Drehzahlregelung 23
3.3.5 Kennlinienaufnahme 23
3.3.6 DC/DC-Wandler 25
3.3.7 Test 25

4 Abschluss 26

4.1 Zusammenfassung 26

4.2 Ausblick 26

Literatur 27

Anlagen 29

Anhang 1: Schaltplan I

Anhang 2: CAPL Skript ECU_sim.can II

Anhang 3: CAPL Skript BMS_sim.can IV

Anhang 4: CAPL Skript Init.cin V

Anhang 5: CAPL Skript Functions.cin VII

Anhang 6: CAPL Skript Test.can IX

Anhang 7: Kurzanleitung Kennlinienaufnahme XI

Selbstständigkeitserklärung 43

Abbildungsverzeichnis

Abbildung 1: Teststand Gesamtansicht 2

Abbildung 2: Inverter 3

Abbildung 3: Motor 3

Abbildung 4: DC-Netzgerät 5

Abbildung 5: Relais 6

Abbildung 6: CAN-Interface 8

Abbildung 7: Schaltplan 9

Abbildung 8: Stecker 9

Abbildung 9: Verteiler 10

Abbildung 10: Batterie 10

Abbildung 11: Batterieschalter 11

Abbildung 12: Erdung 11

Abbildung 13: Kühlkreislauf 12

Abbildung 14: Kühlmittelpumpe 12

Abbildung 15: CANoe Oberfläche 13

Abbildung 16: Knotenkonfiguration Beispiel 14

Abbildung 17: Graphics-Fenster 15

Abbildung 18: Hardware-Konfiguration 15

Abbildung 19: Kanalzuordnung 15

Abbildung 20: Panel Designer Ausschnitt 16

Abbildung 21: Systemvariablen 17

Abbildung 22: GUI Steuerung 18

Abbildung 23: GUI Überwachung 19

Abbildung 24: Nachrichten Initialisierung Beispiel 19

Abbildung 25: Initialisierung ECU_sim.can 20

Abbildung 26: Nachrichten senden Beispiel 20

Abbildung 27: Aktivierung/Deaktivierung des Inverters 21

Abbildung 28: Abschaltbedingungen 22

Abbildung 29: Wechsel der Betriebsart 22

Abbildung 30: Ansteuerung Inverter 22

Abbildung 31: Drehzahlregler 23

Abbildung 32: Kennlinienaufnahme Variablen 23

Abbildung 33: Kennlinienaufnahme Vorbereitungen 24

Abbildung 34: Kennlinienaufnahme Durchführung 24

Abbildung 35: Kennlinienaufnahme Beispieldatei 25

Abbildung 36: Ansteuerung DC/DC-Wandler 25

Abbildung 37: Test.can Ausschnitt 25

Tabellenverzeichnis

Tabelle 1: Daten Maschinensatz 4

Tabelle 2: Daten DC-Netzgerät 6

Tabelle 3: Daten K1 7

Tabelle 4: Daten K2 7

Tabelle 5: Daten CAN-Interface 8

Abkürzungsverzeichnis

Abkürzung	Langfassung	Übersetzung/Bedeutung
BMS	Battery Management System	Batteriesteuerung
CAN	Controller Area Network	Verwendetes Bussystem
CAPL	Communication Access Programming Language	Von Vector Informatik entwickelte Programmiersprache
GUI	Graphic User Interface	Mensch-Maschinen-Schnittstelle
HV	High Voltage	Hochvoltsystem ~400V
LV	Low Voltage	Niederspannungssystem ~12V
PESM	Permanenterregte Synchronmaschine	
TMM	Technikum Mittweida Motorsport	Rennsportteam der Hochschule
VCU	Vehicle Control Unit	Fahrzeug-Steuergerät

1 Einleitung

1.1 Hintergrund und Aufgabenstellung

Kernstück des neuen Teststandes soll ein Maschinensatz der Firma Robert Bosch GmbH bestehend aus einem Motor SMG-180.1.3 und einem Inverter INV-CON.E-2.3 werden. Dieser Satz ist einer von zweien, die zu einem älteren Teststand verkuppelt waren. Dieser diente vor allem zum Einrichten und Ausloten der Ansteuerungsmöglichkeiten der Geräte. Der andere Maschinensatz wurde später im Elektrorennwagen des Motorsportteams der Hochschule TMM verbaut. Die Erfahrungen, die in diesen Projekten gesammelt wurden, sollen nun für den Aufbau eines neuen Teststandes genutzt werden.

Die Hauptaufgabe des neuen Teststandes soll in der Erprobung von Elektromotoren und der Ermittlung ihrer Kennparameter bestehen. In der Zukunft sind auch Belastungstest von Energiespeichern geplant. Um dies zu ermöglichen, muss dem Bosch Maschinensatz eine 300 – 400 V_{DC} Versorgungsspannung und eine 12 V_{DC} Steuerspannung zur Verfügung gestellt werden. Für den bereitgestellten Steuer-PC muss ein GUI entwickelt werden. Diese soll die Kommunikation mit dem Inverter über einen CAN-Bus übernehmen und für den Bediener in einer einfach zu handhabenden graphischen Oberfläche darstellen. Diesem soll es dabei möglich sein, zwischen einer Drehmomentregelung, einer Drehzahlregelung und dem Betrieb einer automatisierten Aufnahme der Drehzahl-Drehmomenten-Kennlinie zu wählen. Des Weiteren muss für die Wasserkühlung des Maschinensatzes ein entsprechender Kühlkreislauf eingerichtet werden.

1.2 Aufbau

Diese Arbeit ist in zwei generelle Teile gegliedert. Der erste Teil wird sich mit dem physischen Aufbau des Teststandes beschäftigen. Die verwendeten Bauteile und -gruppen werden vorgestellt und ihre Funktion und Verwendung beschrieben. Außerdem wird die gesamte Schaltung dargestellt.

Der zweite Teil ist der Software des Teststandes gewidmet. Er enthält eine kurze Vorstellung der verwendeten Programme, eine Darstellung der graphischen Oberfläche, sowie eine ausführliche Erläuterung des selbstgeschriebenen Codes.

2 Hardware

2.1 Übersicht

Abbildung 1: Teststand Gesamtansicht

Der Teststand wurde auf dem Maschinenbett im Raum 7-019 Prüffeld Elektromobilität installiert. Als Gehäuse wurde in ausrangierter Schaltschrank aufgearbeitet. Er beherbergt alle Komponenten, außer dem Motor und dem HV-Aus-Schalter, der vom Bediener mitgeführt werden soll. Sowohl der Schrank als auch der Motor sind fest mit dem Maschinenbett verbunden, sodass dieses die auftretenden Kräfte, vor allem am Motor, aufnehmen kann.

2.2 Maschinensatz

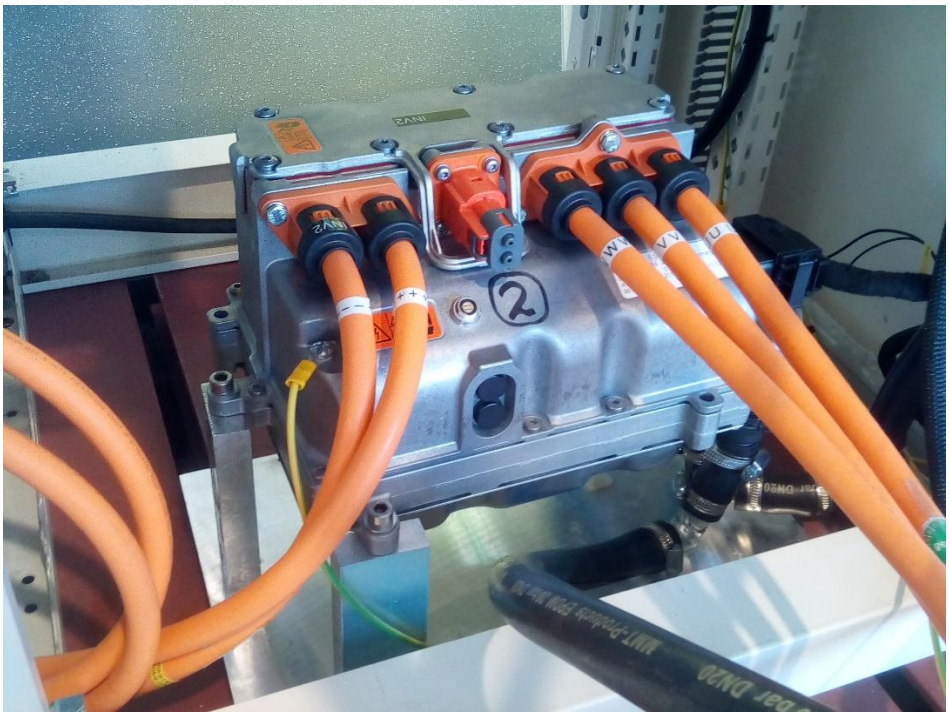


Abbildung 2: Inverter

Der Maschinensatz besteht aus einem Motor SMG-180.1.3 und einem Inverter INV-CON.E-2.3 des Herstellers Robert Bosch GmbH.[1] Da er zuletzt in einem anderen Teststand erfolgreich im Einsatz war, konnte auch ein großer Teil des Zubehörs wiederverwendet werden. Dies betrifft vor allem die eigens gefertigten Halterungen und die Kabel. Insbesondere bei den HV-Kabeln[2] war dies sehr günstig, da es vor Ort keine Möglichkeit gibt, sie zu bearbeiten. Entwickelt wurde dieser Maschinensatz für den Einsatz in Elektrofahrzeugen, was sich teilweise in der Bezeichnung von Anschlüssen und CAN-Nachrichten wiederspiegelt.

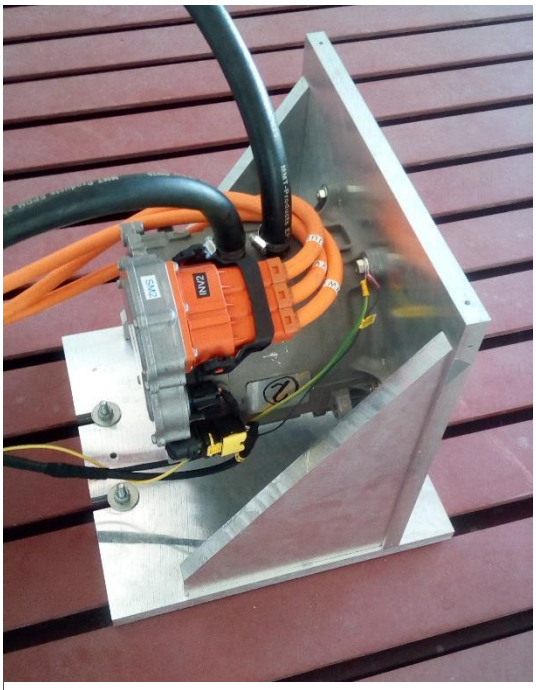


Abbildung 3: Motor

[1] [Bosch]

[2] Coroplast FHLR2GCB2G 50 mm² Cu geschirmt

Neben dem gesteuerten Wechselrichter zur Versorgung des Motors enthält der Inverter auch einen DC/DC-Wandler. Dieser wird genutzt, um die Steuerspannung zu erzeugen.

Eigenschaft	**Wert**
Eingangsspannung	150 – 400 V_{DC}
Steuerspannung	10,6 – 15 V_{DC}
Maximales Drehmoment	200 Nm
Maximale Drehzahl	12.800 min^{-1}
Maximale Leistung	90,5 kW
Temperaturbereich Motor **Inverter**	Max 85 °C -30 – 65 °C

Tabelle 1: Daten Maschinensatz

2.3 DC-Netzgerät

Abbildung 4: DC-Netzgerät

Um die Eingangsspannung für den Inverter zur Verfügung zu stellen, wurde ein Netzgerät PSI 9500-303U der Firma dataTec angeschafft.[3] Es kann Spannungs-, Strom- oder Leistungsgeregelt betrieben werden und verfügt über einen Funktionsgenerator, der sowohl diverse Standardfunktionen (Sinus, Trapez, Rampe) als auch frei erstellbare Sequenzen erzeugen kann. Zum Schutz vor Überhitzung ist das Gerät mit einer eigenen, temperaturgeregelten Kühlung ausgestattet. Außerdem besteht die Möglichkeit mehrere Geräte über einen Master-Slave-Bus zu verbinden, oder das Gerät über USB oder Analogschnittstelle fernzusteuern.

Im Teststand soll das Gerät als Konstantspannungsquelle mit U_{DC} = 400 V dienen. Seine maximalen Werte für Leistung und Strom bilden aktuell die Grenzen der Gesamtperformance des Teststandes.

[3] [dataTec]

Eigenschaft	Wert
Eingangsspannung	400 V_{AC}
Eingangsstrom	2 x 16 A (L2 und L3)
Maximale Spannung	500 V_{DC}
Maximaler Strom	30 A
Maximale Leistung	5 kW
Temperaturbereich	0 – 50 °C

Tabelle 2: Daten DC-Netzgerät

2.4 Relais

Abbildung 5: Relais

Die Schaltung enthält zwei zusätzliche Relais, beide vom Hersteller TE Connectivity.

K1 ist ein zweipoliges HV-Relais EV202AVAND und dient zur Unterbrechung der HV-Spannung am Eingang des Inverters. Angesteuert vom HV-Aus-Schalter kann so ein sicheres Abschalten des Maschinensatzes, auch während des Betriebes, gewährleistet werden.

Eigenschaft	**Wert**
Spulenspannung	12 V_{DC}
Spulenstrom (Anzug / Halten)	5 A / 0,6 A
Maximale Kontaktspannung	500 V_{DC}
Nennkontaktstrom	350 A
Maximaler Abschaltstrom	700 A

Tabelle 3: Daten K1

K2 ist ein Standard-Leiterplattenrelais OZ-SS-112LM1F. Seine Aufgabe ist es den Steuereingang „KL15“ des Inverters zu schalten. Dieser entspricht der Zündung in einem KFZ. Über ihn ist es möglich den Inverter zu aktivieren und zu deaktivieren, ohne die Steuerspannung zu unterbrechen zu müssen.

Eigenschaft	**Wert**
Spulenspannung	12 V_{DC}
Spulenstrom	45 mA
Maximale Kontaktspannung	24 V_{DC}
Nennkontaktstrom	16 A
Maximaler Abschaltstrom	16 A

Tabelle 4: Daten K2

2.5 CAN-Interface

Abbildung 6: CAN-Interface

Da es sich in den vorangegangenen Projekten bewährt hat, soll auch in diesem ein VN1630A CAN-Interface der Firma Vector Informatik GmbH[4] die Kommunikation zwischen PC und Inverter übernehmen. Dieses Modul verfügt über zwei CAN-Kanäle, zwei Steckplätze für weitere Kanäle, zwei digitale Eingänge, einen analogen Eingang und einen digitalen Ausgang. Die Verbindung zu PC erfolgt über USB.

Eigenschaft	**Wert**
Spannungsversorgung	5 V (über USB)
Leistungsaufnahme	2,5 W
CAN-Protokoll	CAN High-Speed 1051cap
CAN-Geschwindigkeit	Bis zu 2 Mbit/s

Tabelle 5: Daten CAN-Interface

[4] [Vector]

2.6 Schaltung

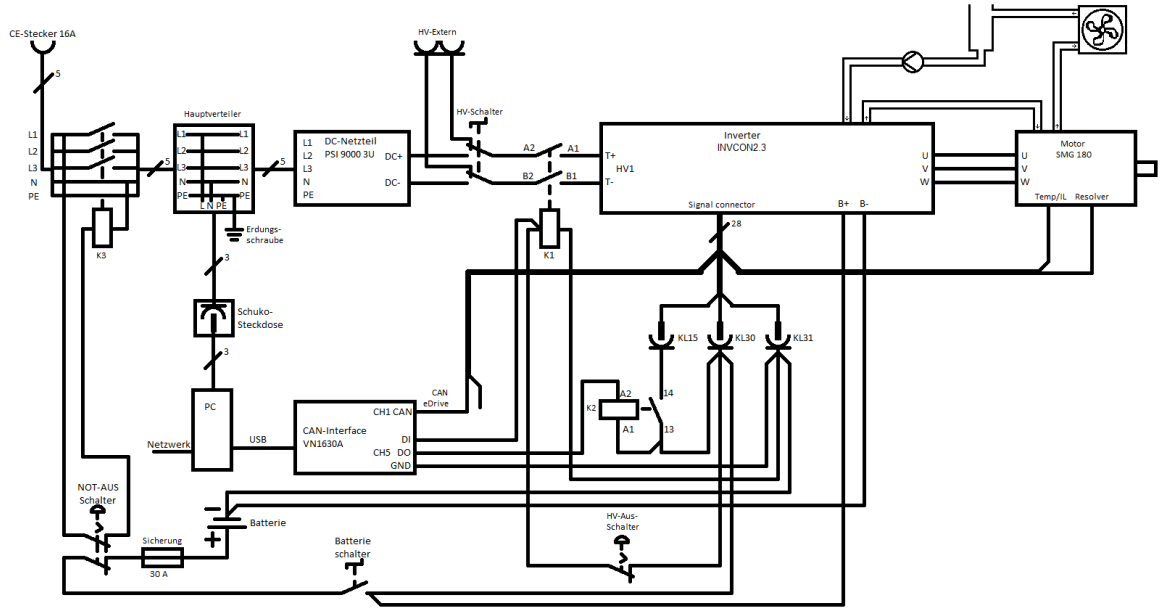


Abbildung 7: Schaltplan[5]

Die Anlage wird durch einen 16 A CE-Stecker (siehe Abbildung 8) versorgt. Da die entsprechende Steckdose leider nicht von den NOT-AUS-Schaltern des Raumes abgeschalten werden kann, muss für den Teststand ein eigener NOT-Aus mit Schalter und Relais (K3) vorgesehen werden. Der NOT-AUS-Schalter ist mit zwei Öffnern versehen, um im Notfall sowohl Versorgungs- als auch Steuerspannung zu trennen und so die gesamte Anlage stillegen zu können. Niemals darf der NOT-AUS-Schalter jedoch zum Abschalten der Anlage verwendet werden! Die im HV-Kreis des Inverters gespeicherte Energie kann diesen zerstören, wenn die Verbindung zur Steuerspannung unterbrochen wird.

Abbildung 8: Stecker

[5] Vergrößerte Version im Anhang 1

Abbildung 9: Verteiler

Nach dem NOT-AUS-Relais wird die Zuleitung im Hauptverteiler aufgeteilt. Der Anschluss des DC-Netzteils ist allphasig ausgeführt, auch wenn nur die Phasen L2 und L3 belastet werden. Daher ist die schaltbare Steckdosenleiste an der Phase L1 angeschlossen. Sie bietet 6 Steckplätze für den PC, Bildschirme und ähnliche Geräte.

Zwischen dem Netzteil und dem HV-Relais (K1) befindet sich ein zweipoliger Umschalter, der es ermöglichen soll, den Inverter alternativ auch von einer externen Quelle zu versorgen.

In diesem Bereich der HV-Spannung wurde NASGFÖU Gummi-Aderleitung mit einem Querschnitt von 16 mm² verlegt. Diese Leitung hat eine Nennspannung U_0 von 1,8 kV und ist für die maximalen 30A des DC-Netzteils mehr als ausreichend dimensioniert. Vom HV-Relais zum Inverter und von dort bis zum Motor wurde die vorkonfektionierte Leitung von Bosch verwendet.

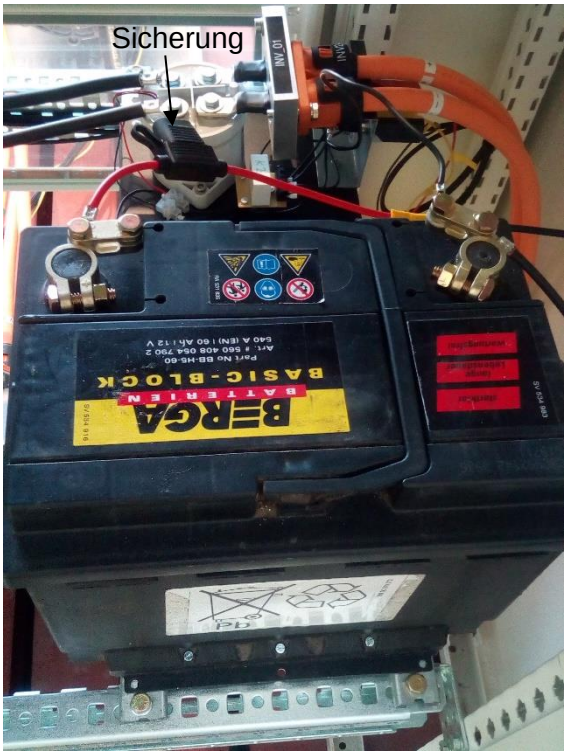


Abbildung 10: Batterie

Die Steuerspannung von etwa 12 V wird von einer Berga BB-H5-60[6] Autobatterie gestützt. Sie verfügt über eine Kapazität 60 Ah, einen Kaltstartstrom von 540 A und ist für eine Ladespannung bis 14,4 V ausgelegt. Als Überstromschutz ist eine 30 A Schmelzsicherung mit einem In-Line Sicherungshalter verbaut. Ein Batterieschalter (siehe Abbildung 11) dient als allgemeiner Zuschaltpunkt für die Steuerspannung. Auch hier gilt das die Steuerspannung niemals vor der HV-Spannung abgeschalten werden darf! Es besteht sonst die Gefahr, das elektronische Komponenten im Inverter zerstört werden.

[6] [Berga]

Abbildung 11: Batterieschalter

Die Anschlüsse des Inverters und des Motors sind durch einen vorgefertigten Kabelbaum verbunden, der vom alten Teststand wiederverwendet werden konnte. Die Ausgänge dieses Kabelbaumes bilden ein 9-poliger Sub-D Stecker für den CAN-Bus, der direkt mit dem CAN-Interface verbunden wird, und drei Bananenstecker für die Eingänge „KL15“ (Zündung), „KL30“ (Steuerspannung) und „KL31“ (Masse). Für diese Stecker wurden drei entsprechende Buchsen installiert, die gleichzeitig als Verteiler für die Steuerspannung dienen. So werden von hier die Relais K1 und K2, der HV-Aus Schalter und der digitale Ausgang des CAN-Interfaces mit 12 V bzw. Masse versorgt. Zu guter Letzt ist der digitale Eingang des CAN-Interfaces mit dem Eingang des HV-Relais verbunden. Dadurch ist es der Software möglich, den Zustand des Relais als Variable zu verwenden.

2.6.1 Erdung

Abbildung 12: Erdung

Die Erdung der meisten Bauteile ist durch das Metall des Schaltschrankes gegeben, dessen Einzelteile großflächig mit einander verschraubt oder vernietet sind. An der zentralen Erdungsschraube unterhalb des Hauptverteilers kann das Erdpotential abgegriffen werden. Dort laufen auch die separaten Erdungsleitungen für Motor und Inverter und die Verbindung zum PE-Leiter der Zuleitung zusammen. Die Schirme der Leitungen für HV-Spannungen und den Motor-Resolver sind über Gehäuseschrauben geerdet. Abgesehen von Schirmen und des Neutralleiters der Zuleitung ist kein weiteres Potential, ob HV oder LV, mit dem Erdpotential verbunden.

2.6.2 Kühlung

Abbildung 13: Kühlkreislauf

Der Kühlkreislauf wurde beinahe vollständig vom alten Teststand übernommen. Am aktuellen Aufbau gemessen ist er eigentlich deutlich überdimensioniert. Erstens ist er ursprünglich auf zwei der verwendeten Maschinensätze ausgelegt. Zweitens stellt das DC-Netzgerät als aktuelle Quelle nur eine Leistung von 5 kW zur Verfügung, die überhaupt in Wärme umgewandelt werden könnte.

Der Kühlkreislauf besteht, neben Schläuchen, aus einer Pumpe, einem Kühler und einem Ausgleichsbehälter. Gekühlt werden nur Motor und Inverter, alle anderen Geräte verfügen, wenn nötig, über eine eigene Lüfter-getriebene Kühlung. Als Kühlmittel wird Wasser verwendet. Der Einsatz von Glycerin-haltigem Si-OAT ist möglich.[7] Das Kühlmittel wird erst durch den Inverter und dann durch den Motor geleitet, um den unterschiedlichen Maximaltemperaturen der Geräte gerecht zu werden. Um ein Trockenlaufen zu vermeiden, befindet sich die Pumpe am niedrigsten Punkt des Kühlkreislaufs.

Abbildung 14: Kühlmittelpumpe

[7] [Bosch]

3 Software

3.1 CANoe

CANoe ist eine Software der Vector Informatik GmbH und bietet Möglichkeiten zur Entwicklung und Prüfung von CAN-Netzwerken. Mit zusätzlichen Optionen kann seine Funktionalität auch auf etliche weitere Netzwerke aus den Bereichen Automotive, Sensorik und Aerospace erweitert werden. Ein großer Vorteil liegt in seinem modularen Aufbau. Es umfasst eine große Anzahl unterschiedlicher Simulations- und Analyse-werkzeuge, die alle aktivierbar und konfigurierbar sind.

Sämtliche während einer Messung gesammelte Daten können vollautomatisch gespeichert, ausgewertet und/oder graphisch dargestellt werden. Messungen können an realen Netzwerken, Simulationen und beliebigen Kombinationen derselben durchgeführt werden. Außerdem existiert ein Offline-Modus, mit dem gespeicherte Messungen erneut wiedergegeben werden können.

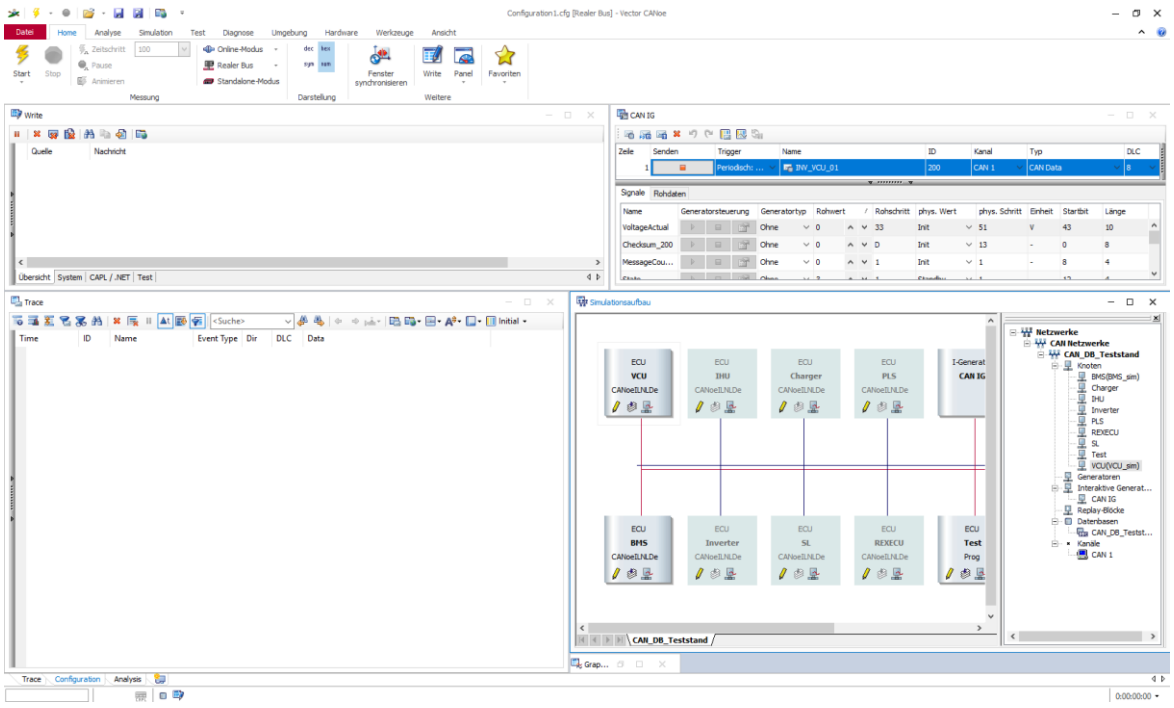


Abbildung 15: CANoe Oberfläche

In dieser Konfiguration sind die folgenden Module sichtbar (im Uhrzeigersinn beginnend unten rechts):

Der Simulationsaufbau:
Eine grafische Darstellung aller angeschlossenen oder simulierten Netzwerke. Die vorhandenen Knoten werden, wie auch die Nachrichten und Signale, aus einer verknüpften CAN-Datenbase übernommen. In der Konfiguration der einzelnen Knoten kann ausgewählt werden, ob ein Knoten simuliert oder nur beobachtet werden soll. Im Fall einer Simulation kann diese als Skript oder Matlab Simulink®-Modell hinterlegt werden. Auch können dem Netzwerk hier weitere Knoten mit rein simulativer Funktion, wie Test und der interaktive Generator, hinzugefügt werden.

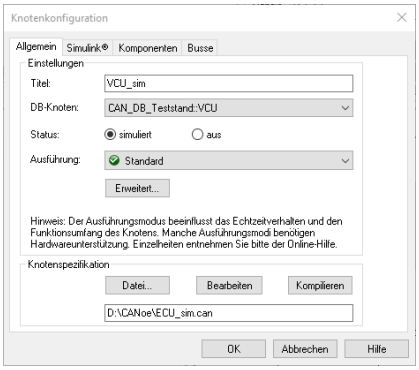


Abbildung 16: Knotenkonfiguration Beispiel

Das Trace-Fenster:
Hier werden während einer Messung alle empfangenen und gesendeten Nachrichten angezeigt. Ist eine Nachricht in der verbundenen Datenbase hinterlegt, stehen neben ihren rohen Daten auch die enthaltenen Signale mit ihren realen Werten[8] zur Verfügung. Alle Nachrichten sind mit einem Zeitstempel versehen, der alternativ auch als Differenz angezeigt werden kann, was die Überprüfung von Zykluszeiten ermöglicht. Die Anzeige kann umgeschaltet werden zwischen der kontinuierlichen Darstellung aller Nachrichten, was insbesondere in Kombination mit Filtern hilft Änderungen im zeitlichen Verlauf zu erkennen, und der aktualisierenden Darstellung der letzten Nachricht jeder ID, was einen schönen Überblick über die aktuellen Werte verschafft. Mit diesen Möglichkeiten ist das Trace-Fenster das wichtigste Werkzeug bei der Fehlersuche in einem Netzwerk.

Das Write-Fenster:
Dies ist eine einfache Textausgabe. Sie informiert über so grundlegende Dinge wie Start, Ende und Dauer der aktuellen Messung, aufgetretene Fehler und andere System-meldungen. Da auch selbstgeschriebene Skripte mit der Funktion write() hier Ausgaben machen können, ist es wichtiges Werkzeug zur Selbstkontrolle beim Programmieren.

Der interaktive Generator CAN IG:
Interaktive Generatoren können während einer Messung frei konfigurierbare Nachrichten in das aktuelle Netzwerk senden. Dies kann entweder periodisch, manuell oder als Reaktion auf eine bestimmte Nachricht oder Variable geschehen. Für die Daten der Nachricht können feste Werte oder definierbare Funktionen eingestellt werden. In diesem Fall wurde er verwendet, um die Funktion der Drehzahlregelung zu testen.

[8] In CAN-Datenbasen können Signale mit Wertetabellen oder Umrechnungsfaktoren hinterlegt werden, um z.B. die Übertragung gebrochener Zahlen oder nichtnumerischen Werte zu ermöglichen.

Aktuell minimiert ist das Graphics-Fenster:
Hier kann man sich Signale in einem Diagramm darstellen lassen. Als X-Achse kann dabei entweder die Zeit oder, wie in diesem Fall, ein anderes Signal gewählt werden. Leider konnte es nicht zur Kennlinienaufnahme genutzt werden, da sich Anfang, Ende und Speicherung der Aufnahme nicht automatisieren lassen.

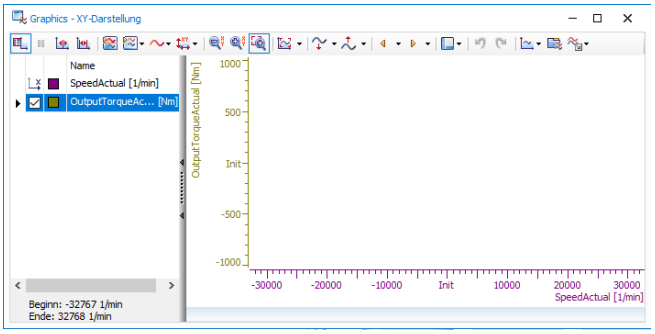


Abbildung 17: Graphics-Fenster

3.1.1 Hardware-Verknüpfung

Wenn das CAN-Interface mit dem Rechner verbunden ist, wird die Verbindung von CANoe normalerweise automatisch hergestellt. Im Tab „Hardware“ kann das Interface dann konfiguriert werden. Dabei ist es wichtig auf die korrekte Baudrate zu achten. Der Inverter fordert eine Verbindung mit 500 kBaud.

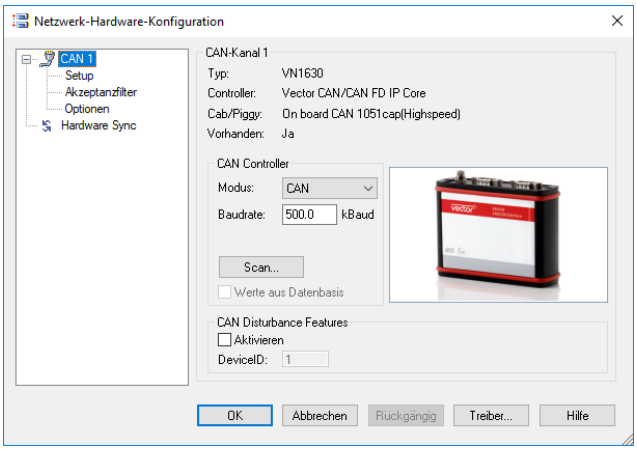


Abbildung 18: Hardware-Konfiguration

Außerdem sollte man die Zuordnung der Kanäle prüfen. Hier stehen auch virtuelle Kanäle zur Verfügung, mit denen z.B. die Skripte in einer sicheren Umgebung getestet werden können.

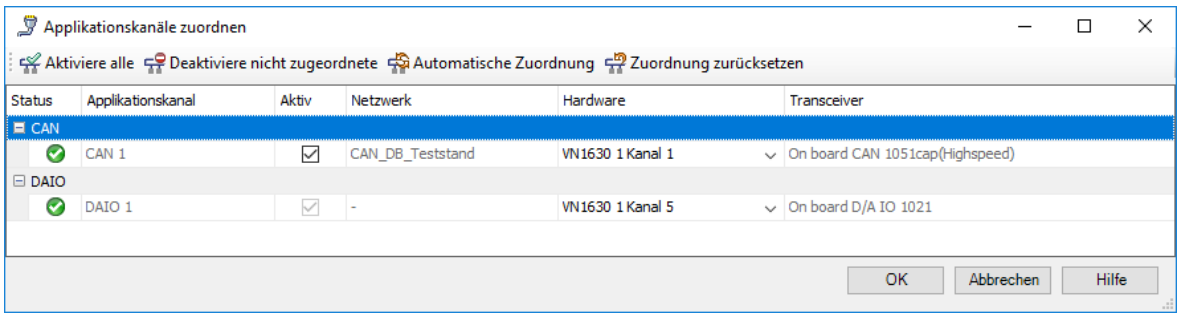


Abbildung 19: Kanalzuordnung

3.1.2 Werkzeuge

CANoe stellt eine Reihe von weiteren Programmen zur Verfügung, die Werkzeuge genannt werden. Die wichtigsten darunter sind der CANdb++ Editor, der Panel Designer und der CAPL Browser.

Der CANdb++ Editor dient dem Erstellen und Bearbeiten von CAN-Datenbasen. Er soll hier nicht näher beschrieben werden, da für dieses Projekt die zum Maschinensatz mitgelieferte Datenbase der Firma Bosch verwendet wurde.[9]

3.1.2.1 Panel Designer

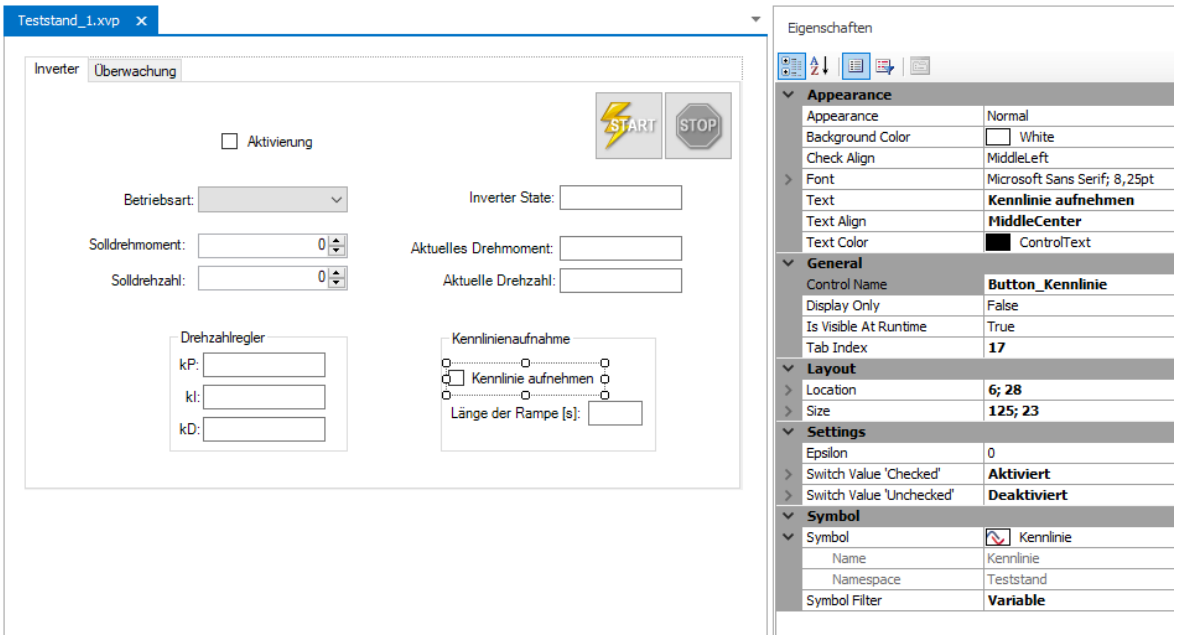


Abbildung 20: Panel Designer Ausschnitt

Der Panel Designer ermöglicht das Erstellen eigener Fenster. Diese können mit diversen Formen von Kontroll- und Anzeigeelementen wie Schaltern, Skalen und Auswahlfeldern gefüllt werden. Jedes dieser Elemente verfügt über eine Anzahl Eigenschaften, die sein Aussehen und seine Funktion beschreiben. Diese Eigenschaften können auch während einer Messung noch durch CAPL-Funktionen verändert werden, um beispielsweise die Sichtbarkeit oder Bedienbarkeit eines Elementes zu ändern. Des Weiteren lässt sich jedem Element eine Systemvariable oder ein CAN-Signal zuordnen, das von diesem Element dargestellt und/oder beeinflusst wird.

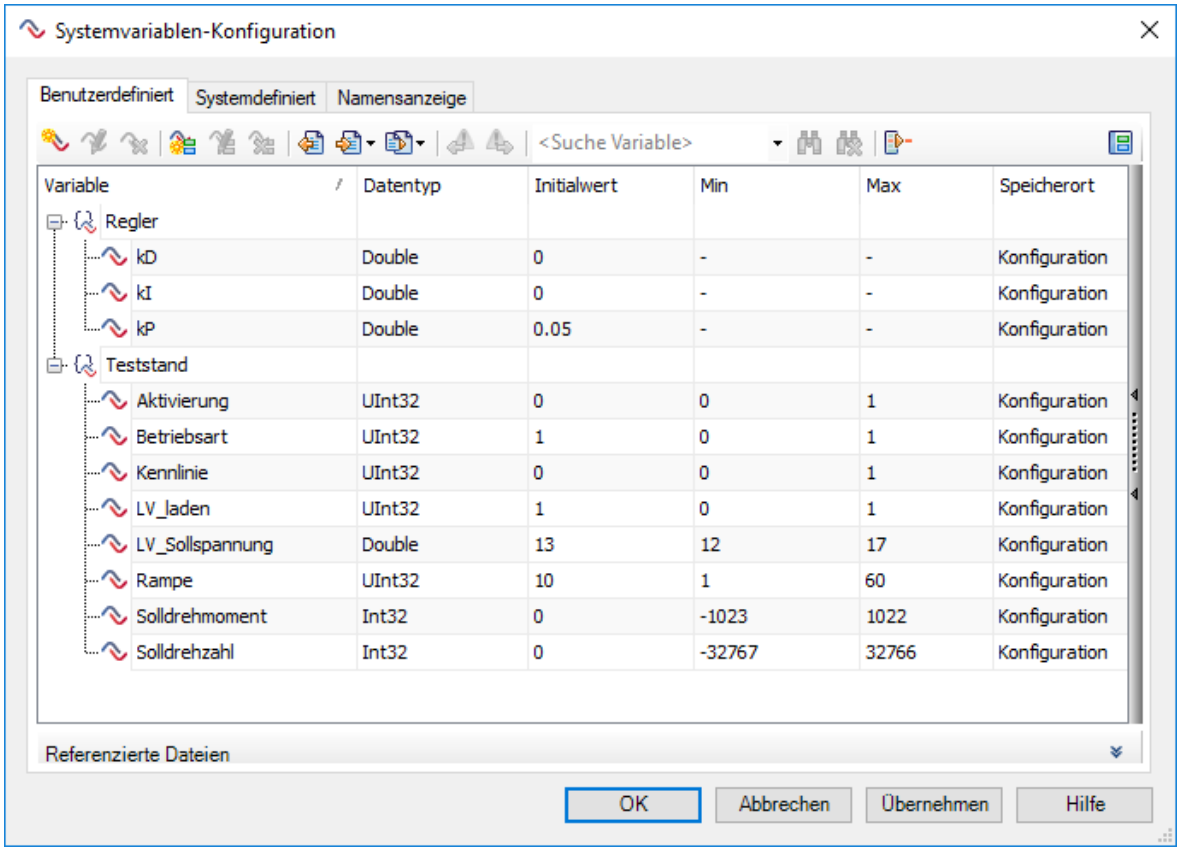
[9] Anwendungsbeispiel: [Donath]

3.1.2.2 CAPL Browser[10]

Mit dem CAPL Browser können Skripte in der Vector Informatik entwickelten Programmiersprache CAPL erstellt und kompiliert werden. Diese ist eng an C angelehnt und enthält „eine Vielzahl von vordefinierten Funktionen, die das Arbeiten mit dem Entwicklungs-, Test- und Simulations-Werkzeug CANoe und CANalyzer unterstützen.“ Außerdem sind sämtliche Signale und Nachrichten aus der CAN-Datenbase sowie die Systemvariablen als Objekte verfügbar. CAPL-Skripte werden ereignisbasiert ausgeführt, d.h. beim Eintreten eines bestimmten Ereignisses wird die diesem Ereignis zugeordnete Routine ausgeführt. Ereignisse sind beispielsweise Messungsstart und –ende, das Auslaufen eines Timers, das Eintreffen einer Nachricht oder die Veränderung einer Systemvariable. Pointer werden von CAPL nicht unterstützt.

3.1.3 Systemvariablen

Zur Kommunikation zwischen den Skripten und dem Panel werden sogenannte Systemvariablen benötigt. Diese werden in CANoe unter dem Tab „Umgebung“ definiert.



Variable	Datentyp	Initialwert	Min	Max	Speicherort
Regler					
kD	Double	0	-	-	Konfiguration
kI	Double	0	-	-	Konfiguration
kP	Double	0.05	-	-	Konfiguration
Teststand					
Aktivierung	UInt32	0	0	1	Konfiguration
Betriebsart	UInt32	1	0	1	Konfiguration
Kennlinie	UInt32	0	0	1	Konfiguration
LV_laden	UInt32	1	0	1	Konfiguration
LV_Sollspannung	Double	13	12	17	Konfiguration
Rampe	UInt32	10	1	60	Konfiguration
Solldrehmoment	Int32	0	-1023	1022	Konfiguration
Solldrehzahl	Int32	0	-32767	32766	Konfiguration

Abbildung 21: Systemvariablen

[10] [CAPL]

Die Systemvariablen wurden in zwei Namensräumen angelegt. „Regler“ enthält die Parameter des Drehzahlreglers. Sollte man diese berechnen, ist zu beachten, dass es sich hierbei um einen diskreten Regler mit einer Schrittweite von 10 ms handelt. Die im Namensraum „Teststand“ enthaltenen Variablen dienen zur Ansteuerung des Systems. „Aktivierung“ aktiviert den Inverter, „Betriebsart“ schaltet zwischen Drehzahl- und Drehmomentregelung um, „Kennlinie“ startet die Aufnahme der Kennlinie und „LV_Laden“ erlaubt es dem DC/DC-Wandler, die Steuerspannung bereit zu stellen. Bei „LV_Sollspannung“, „Solldrehmoment“ und „Solldrehzahl“ handelt es sich um die sollwerte der entsprechenden Größen. „Rampe“ schließlich ist der Richtwert, über wie viele Sekunden die Kennlinie abgefahren werden soll.

3.2 GUI

Als GUI wurde im Panel Designer das Fenster „Teststand_1“ mit zwei Tabs erstellt. Beide enthalten einen START/STOP-Schalter, der eine Messung beginnt bzw. beendet.

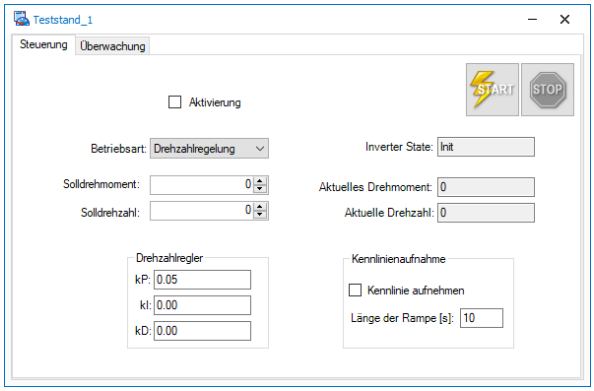


Abbildung 22: GUI Steuerung

Der erste Tab enthält alle nötigen Elemente zur Steuerung des Teststandes. Der Schalter „Aktivierung“ aktiviert den Inverter und wechselt so von Leerlauf in angetriebenen Modus. Darunter lässt sich nun die gewünschte Betriebsart, Drehzahl- oder Drehmomentregelung auswählen. Drehzahlregelung ist der Standardwert. Je nach Betriebsart seht nun eine der beiden Sollwerteingaben zur Verfügung. Der Sollwert kann entweder direkt in das Feld eingegeben oder mit den Pfeiltasten inkrementell verändert werden. Rechts neben den Sollwerten werden die aktuellen Werte angezeigt. Bei einem Wechsel der Betriebsart wird immer der aktuelle Wert als neuer Sollwert übernommen, um unvorhersehbare Sprünge zu verhindern. Im unteren Bereich besteht auf der linken Seite die Möglichkeit, die Parameter des Drehzahlreglers an zu passen. Dies sollte bereits vor der Aktivierung des Inverters passieren, ist aber auch im laufenden Betrieb möglich. Auf der rechten Seite befindet sich die Steuerung der automatischen Kennlinienaufnahme. Der Schalter „Kennlinie aufnehmen“ startet den Vorgang. Er ist nur im Betriebsmodus Drehzahlregelung verfügbar. Die Dauer des Vorgangs lässt sich jederzeit ändern, außer die Kennlinienaufnahme ist aktuell aktiv.

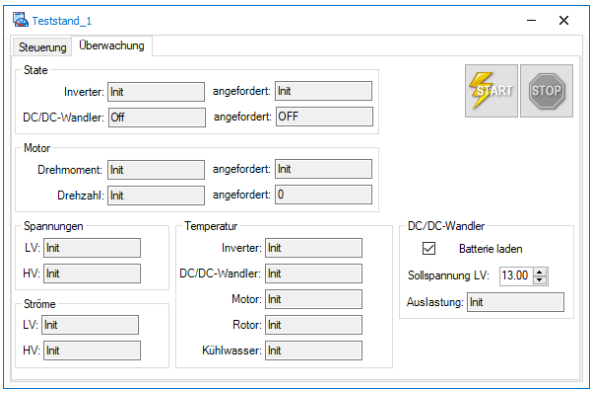


Abbildung 23: GUI Überwachung

Der andere Tab dient der Überwachung des Systems. Hier befinden sich Anzeigen für die internen Betriebsmodi von Inverter und DC/DC-Wandler, Drehmoment und Drehzahl, sowie die von der Steuerung angeforderten Werte für diese Größen. Außerdem werden die Spannungen und Ströme im HV- und LV-Bereich sowie diverse Temperaturen angezeigt. Ebenfalls auf diesem Tab ist die Ansteuerung des DC/DC-Wandlers zu finden. Im Ausgangszustand wird die Steuerspannung automatisch vom DC/DC-Wandler zur verfügung gestellt, sobald eine ausreichende HV-Spannung anliegt. Dieser Automatismus kann mit dem Schalter „Batterie laden“ aus- bzw. wieder eingeschalten werden. Im Feld „Sollspannung LV“ kann der Wert dieser Spannung geändert werden. Zu beachten ist noch, dass der Zugriff des Inverters eine höhere Priorität hat. Zum Beispiel wenn dieser den DC/DC-Wandler nutzt, um den HV-Kreis zu entladen.

3.3 CAPL Skripte

Die CAPL-Skripte sind aufgeteilt in 5 Dateien: ECU_sim.can als Simulationsobjekt des Knotens VCU enthält die diversen Steuerungsaufgaben, BMS_sim.can als Simulationsobjekt des Knotens BMS übernimmt das periodische Senden der meisten Nachrichten, Init.cin gibt für alle Signale die Standardwerte vor, Functions.cin stellt grundlegende Funktionen zur Berechnung von Message Countern und Checksummen zur Verfügung und Test.can ist schließlich ein kleines Skript zur Selbstkontrolle.

Zur besseren Nachvollziehbarkeit ist die Erläuterung nach Funktionen geordnet. Dargestellte Codeabschnitte stellen immer nur Auszüge dar. Eine vollständige Darstellung der Skripte findet sich im Anhang.

3.3.1 Systemstart

```
142 void init_VCU_DC_01(void)
143 {
144   // source: VCU | sink: Inverter
145   VCU2DC_01.MessageCounter_130 = 0x00;
146   VCU2DC_01.VoltageDemandHV.phys = 0;
147   VCU2DC_01.CurrentDemandHV.phys = 0;
148   VCU2DC_01.CurrentDemandLV.phys = 0;
149   VCU2DC_01.VoltageDemandLV.phys = 14;
150   VCU2DC_01.ModeRequest = vtSig_ModeRequest::OFF;
151
152   VCU2DC_01.Checksum_130 = calc_Checksum_130();
153
154   output(VCU2DC_01);
155 }
```

Abbildung 24: Nachrichten Initialisierung Beispiel

Mit dem Beginn einer neuen Messung wird von allen Skripten die Routine `on start` ausgeführt. In dieser wird an erster Stelle die Initialisierung der Nachrichten aufgerufen. ECU_sim.can ist dabei für `init_VCU_DC_01();` und `init_VCU_INV_04();` zuständig, da es diese Nachrichten bearbeitet. BMS_sim.can übernimmt die übrigen zu sendenden Nachrichten. Die aufgerufenen Funktionen befinden sich im Skript Init.cin. Darin werden die Signale der

jeweiligen Nachricht mit ihren Standardwerten versehen, ein eventueller Message Counter und Checksumme initialisiert und die Nachricht einmalig gesendet.

Anschließend werden die Timer für die zyklischen Aufgaben gestellt. Dies geschieht mit jeweils zwei Befehlen, hier am Beispiel von ECU_sim.can: `timer_10_1.cancel();` löscht eventuell noch vorhandene Timer, die z.B. aus vorherigen Messungen stammen könnten und `setTimerCyclic(timer_10_1, 5, 10);` setzt einen neuen Timer auf. Dabei steht die 5 für die Zeit in ms, bis der Timer das erste Mal auslöst und die 10 für die Zyklusdauer in ms nach der sich der Timer erneut meldet. Der Timer `timer_10_1` löst also bei 5 ms aus und dann bei 15 ms, 25 ms, 35 ms, usw. Bei jeder Auslösung wird die entsprechende `on timer` Routine ausgeführt. Das Skript BMS_sim.can besitzt eigene Timer mit den Zyklen 10 ms, 20 ms, 100, ms und 500 ms.

Als letzte Initialisierungsschritte muss ECU_sim.can noch mit `setWritePath ("D:\\Logging Dateien");` den Speicherort der Kennlinien festlegen und die noch nicht benötigten Elemente des Panels unzugänglich machen.

```
97  on start
98  {
99    init_VCU_DC_01();                                          //Initialisierung der Nachrichten
100   init_VCU_INV_04();
101
102   timer_10_1.cancel();                                       //Initialisierung des Timers
103   setTimerCyclic(timer_10_1, 5, 10);
104   setWritePath("D:\\Logging Dateien");                       //Initialisierung des Dateipfades
105   write("Initialisierung VCU durchgeführt");
106
107   enableControl("Teststand_1", "Auswahl_Betriebsart", 0);    //Initialisierung der Panelelemente
108   enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);
109   enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
110   enableControl("Teststand_1", "Button_Kennlinie", 0);
111   write("Initialisierung Panel durchgeführt");
112 }
```

Abbildung 25: Initialisierung ECU_sim.can

3.3.2 Nachrichten

Alle Nachrichten an den Inverter müssen in festgelegten Intervallen gesendet werden. Dafür wurden die Timer mit genau diesen Zyklen angelegt. Für das Senden einer Nachricht existiert die Funktion `output();`. Die Signale verhalten sich dabei wie Variablen, d.h. sie können zu jedem Zeitpunkt geändert werden und ihr Wert wird im Augenblick des Sendens übernommen. Sollte eine Nachricht einen Message Counter und/oder eine Checksumme enthalten, so muss diese natürlich vor dem Senden aktualisiert werden.

Beispiel au BSM_sim.can: Beide Nachrichten haben einen Zyklus von 500 ms. BMS_VCU_INV_05 verfügt über Message Counter und Checksumme, BMS_INV_01_GW nicht.

```
49  on timer timer_500_2
50  {
51    BMS2VCUINV_05.MessageCounter_96 = set_messageCounter(BMS2VCUINV_05.id);
52    BMS2VCUINV_05.Checksum_96 = calc_Checksum_96();
53    output(BMS2VCUINV_05);
54    output(BCM2INV_01_GW);
55  }
```

Abbildung 26: Nachrichten senden Beispiel

3.3.3 Aktivierung / Betriebsart

```
on sysvar Teststand::Aktivierung
{
  if (@this == 1)
  {
    if (@IO::VN1600_1::DIN0 == 1 && $State == VtSig_State::Standby)
    {
      @IO::VN1600_1::DOUT = 1;                                  //KL15 einschalten
      VCU2INV_04.RequestedState = VtSig_RequestedState::TrqCtrl;  //Inverter aktivieren
      @Teststand::Betriebsart = 1;                              //Betriebsart Drehzahlregelung
      @Teststand::Solldrehzahl = $SpeedActual.phys;
      enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);  //Panelelemente aktivieren
      enableControl("Teststand_1", "Auswahl_Betriebsart", 1);
      enableControl("Teststand_1", "Button_Kennlinie", 1);
    }
    else
    {
      @Teststand::Aktivierung = 0;
    }
  }
  else
  {
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;  //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                    //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                          //kein Drehmoment mehr anfordern
    @Teststand::Kennlinie = 0;                                  //Kennlinienaufnahme beenden
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);  //Panelelemente deaktivieren
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
    enableControl("Teststand_1", "Auswahl_Betriebsart", 0);
    enableControl("Teststand_1", "Button_Kennlinie", 0);
  }
}
```

Abbildung 27: Aktivierung/Deaktivierung des Inverters

Wenn die GUI versucht den Inverter zu aktivieren, indem sie die Systemvariable „Aktivierung“ auf ‚1‘ setzt, werden zwei Bedingungen geprüft. Erstens darf der HV-Aus-Schalter nicht betätigt sein, der der am Digitaleingang „DIN0“ angeschlossen ist. Zweitens muss sich der Inverter im sicheren Zustand „Standby“ befinden. Ist eine dieser Bedingungen nicht erfüllt, wird die Variable „Aktivierung“ direkt wieder zurück auf ‚0‘ gesetzt, ohne den Zustand des Systems zu verändern. Sind beide Bedingungen erfüllt, wird der Inverter aktiviert. Dazu wird der Inverter über den Digitalausgang eingeschaltet und mit dem Signal „Requested State“ aufgefordert, in den Fahrzustand „TrqCtrl“ zu wechseln. Die Betriebsart wird standardmäßig auf Drehzahlregelung mit der aktuellen Drehzahl als Sollwert eingestellt, damit keine sprunghaften Veränderungen an der Maschine stattfinden. Außerdem werden die Panelelemente zur Änderung der Solldrehzahl oder Betriebsart freigegeben.

Die Deaktivierung des Inverters kann von mehreren Stellen angefordert werden. Zunächst natürlich von der GUI, indem sie die Systemvariable „Aktivierung“ auf ‚0‘ setzt. Dabei wird der Inverter in den Zustand „Standby“ geschickt und über den Digitalausgang abgeschaltet. Für den Fall einer noch aktiven Drehzahlanforderung wird diese sicherheitshalber auf ‚0‘ gesetzt. Außerdem wird die eventuelle Aufnahme einer Kennlinie abgebrochen und die freigegebenen Panelelemente werden wieder gesperrt.

```
on sysvar IO::VN1600_1::DIN0                                    //Eingang HV-Aus
{
  if (@this == 0)
  {
    @Teststand::Aktivierung = 0;
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;  //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                    //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                          //kein Drehmoment mehr anfordern
  }
}
on message INV_VCU_01
{
  if (this.State == VtSig_State::Failure)
  {
    @Teststand::Aktivierung = 0;
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;  //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                    //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                          //kein Drehmoment mehr anfordern
  }
}
```

Abbildung 28: Abschaltbedingungen

Außerdem wird der Inverter deaktiviert, wenn der HV-Aus-Schalter betätigt wird oder der Inverter einen Fehler meldet. Hier wird nicht nur die die Systemvariable „Aktivierung“ auf ‚0‘ gesetzt (mit allem eben genannten Konsequenzen), sondern ebenfalls der Inverter über das Signal „RequestedState“ und den Digitalausgang abgeschaltet sowie die Drehmomentanforderung annulliert.

```
on sysvar Teststand::Betriebsart
{
  if (@this == 0)                                               //Betriebsart Drehmomentregelung
  {
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 1);
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
    enableControl("Teststand_1", "Button_Kennlinie", 0);
    @Teststand::Solldrehmoment = $OutputTorqueActual.phys;
    @Teststand::Solldrehzahl = 0;
  }
  else                                                          //Betriebsart Drehzahlregelung
  {
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);
    enableControl("Teststand_1", "Button_Kennlinie", 1);
    @Teststand::Solldrehmoment = 0;
    @Teststand::Solldrehzahl = $SpeedActual.phys;
    eI = 0;                                                     //Zwischenspeicher Regler leeren
    eD = 0;
  }
}
```

Abbildung 29: Wechsel der Betriebsart

Ein Wechsel der Betriebsart kann nur von der GUI durch eine Änderung der gleichnamigen Systemvariable eingeleitet werden. Der Wert ‚0‘ entspricht dabei der Drehmomentregelung, der Wert ‚1‘ der Drehzahlregelung. Beim Wechsel werden zuerst die Panelelemente umgeschaltet, sodass die Sollwertvorgabe nur für die aktuelle Größe verfügbar ist und die Kennlinienaufnahme nur bei Drehzahlregelung möglich ist. Als Sollwert der zu regelnden Größe wird ihr aktueller Wert übernommen, um auch hier ein sprunghaftes Maschinenverhalten zu vermeiden. Der Sollwert der jeweils anderen Größe wird derweil auf ‚0‘ gesetzt.

```
on timer timer_10_1
{
  //Ansteuerung Inverter
  if (@Teststand::Aktivierung == 1)
  {
    if (@Teststand::Betriebsart == 0)                           //Betriebsart Drehmomentregelung
    {
      VCU2INV_04.DesiredTorque.phys = @Teststand::Solldrehmoment;   //Drehmomentanforderung freigegeben
    }
    else                                                        //Betriebsart Drehzahlregelung
    {
      VCU2INV_04.DesiredTorque.phys = Drehzahlregler(@Teststand::Solldrehzahl); //Aufruf Drehzahlregler
    }
  }
  else
  {
    VCU2INV_04.DesiredTorque.phys = 0;                          //Drehmomentanforderung gesperrt
    VCU2INV_04.DesiredSpeed.phys = 0;                           //Drehzahlanforderung gesperrt
  }
}
```

Abbildung 30: Ansteuerung Inverter

Entsprechend der gewählten Aktivierung und Betriebsart findet im 10 ms Zyklus die Anforderung von Drehmoment oder Drehzahl an den Inverter statt. Ist der Inverter deaktiviert (ist die Systemvariable „Aktivierung“ also ‚0‘), dann sind auch die angeforderten Werte ‚0‘. Ist „Aktivierung“ gleich ‚1‘ entscheidet die Betriebsart. Die Drehmomentreglung übernimmt der Inverter selbst, das Solldrehmoment kann also direkt mit dem Signal „DesieredTorque“ versendet werden. Die Drehzahlregelung ist auf diesem Inverter leider nicht freigegeben. Daher muss die Solldrehzahl zunächst einem Drehzahlregler übergeben werden. Dessen Stellgröße ist wiederum ein Drehmoment, das an den Inverter gesendet werden kann.

3.3.4 Drehzahlregelung

```
variables
{
  //Reglervariablen
  long e;                          //Regeldifferenz
  long eI;                         //Zwischenspeicher Integrator
  long eD;                         //Zwischenspeicher Differenzierer
  double uP;                       //P-Anteil
  double uI;                       //I-Anteil
  double uD;                       //D-Anteil
}
long Drehzahlregler (long Sollwert)
{
  e = Sollwert - $SpeedActual.phys;  //Regeldifferenz bilden
  eI = eI + e;                       //Integration
  uP = e * @Regler::kP;              //P-Regler
  uI = eI * @Regler::kI;             //I-Regler
  uD = (e - eD) * @Regler::kD;       //D-Regler
  eD = e;                            //Subtrahend zwischenspeichern

  return (uP+uI+uD);                 //Regler kombinieren
}
```

Abbildung 31: Drehzahlregler

Als Drehzahlregler wurde ein einstellbarer PID-Regler erstellt, dessen Parameter als Systemvariablen angelegt wurden und somit einfach zu ändern sind. Der Aufbau ist nicht optimiert, sollte dafür aber übersichtlicher sein, da die Regleranteile getrennt voneinander berechnet werden. Der P-Anteil ist eine einfache Multiplikation der Regeldifferenz mit der P-Verstärkung. Da es sich um ein diskretes System handelt, kann die Integration zu einer Summation vereinfacht werden. „eI“ ist also die Summe aller bisher aufgetretenen Regeldifferenzen und wird mit der I-Verstärkung multipliziert. Ebenso lässt sich der Differenzierer durch eine Subtraktion ersetzen. Der D-Anteil wird also aus der Differenz zwischen der aktuellen und der vorhergehenden Regeldifferenz in Multiplikation mit der D-Verstärkung gebildet. Die Summe aus P-, I- und D-Anteil wird dann als Stellwert zurückgegeben.

3.3.5 Kennlinienaufnahme

Die automatische Erfassung einer Kennlinie geschieht durch das Abfahren der Drehzahlwerte vom anfänglich eingestellten Wert bis 0 min^{-1} in Form einer Rampe. Dabei werden im 10 ms Takt die jeweils aktuellen Werte von Drehzahl und Drehmoment tabellarisch in eine Datei geschrieben.

Dieser Vorgang wird gesteuert von der Systemvariable „Kennlinie“. Wenn sich das System in der Betriebsart „Drehzahlregelung“ befindet, kann sie von der GUI nach ‚1‘ geändert werden, um eine neue Kennlinie auf zu nehmen. Dafür wird als erstes die manuelle Änderung der Solldrehzahl und der Rampenlänge gesperrt. Die aktuelle Drehzahl wird dann als Anfangspunkt der Sollwert-Rampe übernommen und für später abgespeichert. Um die eingestellte Rampenlänge zu erreichen, muss nun die Größe der Drehzahlschritte je 10 ms Schritt berechnet werden. Dafür wird erst die eingestellte Rampenlänge in s mit 100 multipliziert, um die Anzahl der zur Verfügung stehenden Schritte zu erhalten. Dann wird die Anfangsdrehzahl durch diese Anzahl geteilt und abgerundet. Entsteht dabei eine Schrittweite die kleiner ist als technisch möglich, so wird eine Schrittweite von 1 (bzw. -1 bei negativen Drehzahlen) eingestellt. Für eine Ausgabe im Write-Fenster wird noch die tatsächliche Länge der Rampe berechnet.

```
variables
{
  dword filehandle = 0;
  char datensatz[64];
  char dateiname[34] = "Test.txt";
  long dt[9];              //Datum und Uhrzeit

  //Variablen Kennlinienaufnahme
  long rampenzeit;         //Anzahl der 10ms-Schritte
  long rampenstep;         //Schrittweite in 1/min pro 10ms-Schritt
  long nenndrehzahl;       //Zwischenspeicher für Ausgangspunkt
  int delta = 5;           //Toleranz für "Stillstand" in 1/min
}
```

Abbildung 32: Kennlinienaufnahme Variablen

```
on sysvar Teststand::Kennlinie
{
  if (@this == 0)                                                    //Ende oder Abbruch
  {
    if (@Teststand::Aktivierung == 1)
    {
      enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);       //Rückkehr zu Drehzahlregelung
      @Teststand::Solldrehzahl = nenndrehzahl;
    }
    enableControl("Teststand_1", "Eingabe_Rampe", 1);
    fileClose(filehandle);
  }
  else                                                               //Beginn der Kennlinienaufnahme
  {
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);         //deaktivieren der Panelelemente
    enableControl("Teststand_1", "Eingabe_Rampe", 0);
    @Teststand::Solldrehzahl = $SpeedActual.phys;
    nenndrehzahl = $SpeedActual.phys;                                //Zwischenspeichern der aktuellen Drehzahl
    rampenzeit = @Teststand::Rampe * 100;
    rampenstep = _floor(@Teststand::Solldrehzahl / rampenzeit);      //Berechnen der Schrittweite
    if ((rampenstep < 1) && (rampenstep > -1))                       //Schrittweite mindestens +-1
    {
      rampenstep = sign(@Teststand::Solldrehzahl);
      write("Rampe zu lang für diese Drehzahl! Neue Rampendauer %g s.", abs(@Teststand::Solldrehzahl / 100.));
    }
    else
    {
      write("Berechnete Rampendauer: %g s.", abs((@Teststand::Solldrehzahl / rampenstep) / 100.));
    }
    getLocalTime(dt);                                                //Vorbereiten der Logging-Datei
    snprintf(dateiname, 34, "Kennlinie %d-%d-%d_%d-%d-%d.txt", (dt[5]+1900), dt[4], dt[3], dt[2], dt[1], dt[0]);
    filehandle = openFileWrite(dateiname, 2);
    if (filehandle != 0)
    {
      snprintf(datensatz, 64, "Drehzahl in 1/min\tDrehmoment in Nm\n");
      filePutString(datensatz, 64, filehandle);
    }
  }
}
```

Abbildung 33: Kennlinienaufnahme Vorbereitungen

Als letzter Teil der Vorbereitungen muss noch eine Datei zum Ablegen der Werte angelegt werden. Dafür wird zunächst ein Dateiname in der Form „Kennlinie Jahr-Monat-Tag_ Stunde-Minute-Sekunde.txt“ erzeugt. An dem während der Initialisierung festgelegten Speicherort wird nun eine Datei mit diesem Namen erstellt und mit Schreibrechten geöffnet. War dies erfolgreich, entsteht ein sogenannter Filehandle, der von folgenden Funktionen zur Identifizierung der Datei verwendet wird. Das Beschreiben der Datei erfolgt in zwei Schritten. Zuerst wird mit der Funktion `snprintf();` eine neue Zeile als Zeichenkette „datensatz“ erzeugt. Dann wird diese von der Funktion `filePutString();` an das Ende der Datei angehangen.

```
on timer timer_10_1
{
  //Kennlinienaufnahme
  if (@Teststand::Kennlinie == 1)                                    //Wenn Kennlinienaufnahme aktiv
  {
    @Teststand::Solldrehzahl = @Teststand::Solldrehzahl - rampenstep;  //Verringere Solldrehzahl
    if (abs(@Teststand::Solldrehzahl) <= abs(rampenstep))              //Begrenze Solldrehzahl auf min 0 1/min
      @Teststand::Solldrehzahl = 0;
    if (abs($SpeedActual.phys) <= delta)                             //Bei Stillstand beende Kennlinienaufnahme
      @Teststand::Kennlinie = 0;
    if (filehandle != 0)
    {
      snprintf(datensatz, 64, "%.0f %.0f \n", $SpeedActual.phys, $OutputTorqueActual.phys);
      filePutString(datensatz, 64, filehandle);
    }
  }
}
```

Abbildung 34: Kennlinienaufnahme Durchführung

Wenn diese Vorbereitungen abgeschlossen sind, wird die Kennlinie im 10 ms Zyklus aufgenommen. Dabei wird die Solldrehzahl in jedem 10 ms Schritt um die berechnete Schrittweite reduziert. Wenn die Solldrehzahl dabei ‚0‘ erreicht wird sie dort festgehalten. Sobald die tatsächliche Drehzahl ebenfalls ‚0‘ +- ein kleines Delta erreicht, wird die Kennlinienaufnahme beendet. Eine letzte Prüfung stellt sicher, dass eine beschreibbare Datei vorhanden ist, bevor die aktuellen Werte dort eingetragen werden.

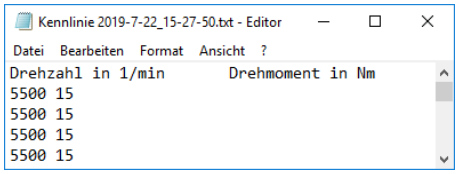


Abbildung 35: Kennlinienaufnahme Beispieldatei

Wird die Systemvariable „Kennlinie“ von der GUI oder einem anderen Teil des Skriptes auf ‚0‘ gesetzt, so wird die aktuelle Kennlinienaufnahme beendet bzw. abgebrochen. Wenn dies nicht mit der Deaktivierung des Inverters verbunden war, kehrt das System zur normalen Drehzahlregelung zurück, indem die Solldrehzahl wieder auf den Wert vor dem Start der Kennlinienaufnahme gebracht und ihre manuelle Eingabe wieder ermöglicht wird. Außerdem wird in jedem Fall die Eingabe einer neuen Rampenzeit freigegeben und die Datei geschlossen.

3.3.6 DC/DC-Wandler

Der DC/DC-Wandler wird aktiviert, wenn die HV-Spannung 300 V übersteigt und dies von der GUI zugelassen wird (was standardmäßig der Fall ist). Die Sollspannung entspricht dabei immer der entsprechenden Systemvariable.

```
on timer timer_10_1
{
  //Ansteuerung DC/DC-Wandler
  if (@Teststand::LV_laden == 1 && $VoltageHV_Actual.phys > 300)
  {
    VCU2DC_01.ModeRequest = VtSig_ModeRequest::Buck;                    //LV-Batterie laden
  }
  else
  {
    VCU2DC_01.ModeRequest = VtSig_ModeRequest::OFF;                     //LV-Batterie nicht laden
  }
  VCU2DC_01.VoltageDemandLV.phys = @Teststand::LV_Sollspannung;         //Spannungsvorgabe DC/DC-Wandler

}
```

Abbildung 36: Ansteuerung DC/DC-Wandler

3.3.7 Test

```
variables
{
  byte MC_BMS_VCU_INV_01 = 0x0F;
  byte MC_BMS_VCU_INV_02 = 0x0F;
  byte MC_BMS_VCU_INV_04 = 0x0F;
  byte MC_BMS_VCU_INV_05 = 0x0F;
  byte MC_BMS_VCU_INV_07 = 0x0F;
  byte MC_PSM_BMS_INV_01_GW = 0x0F;
  byte MC_VCU_DC_01 = 0x0F;
  byte MC_VCU_INV_01 = 0x0F;
  byte MC_VCU_INV_03 = 0x0F;
  byte MC_VCU_INV_04 = 0x0F;
}

on start
{
  write("Initialisierung Test durchgeführt");
}

on message BMS_VCU_INV_01
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6);
  if (temp != 0)
  {
    write("Checksumme ID=6E (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_6E;
  temp = MC_Neu - MC_BMS_VCU_INV_01;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=6E (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_01 = MC_Neu;
}
```

Abbildung 37: Test.can Ausschnitt

Das Skript Test.can wird vom virtuellen Knoten Test ausgeführt und wurde ursprünglich zur Kontrolle der Motorsteuerung des Rennfahrzeugs des TMM geschrieben. Seine Aufgabe besteht darin, bei allen an den Inverter gesendeten Nachrichten Checksumme und Message Counter zu überprüfen. Sollten dabei Fehler festgestellt werden, wird eine entsprechende Meldung im Write-Fenster ausgegeben.

4 Abschluss

4.1 Zusammenfassung

Im Rahmen dieser Masterarbeit ist ein voll funktionsfähiger Motorenteststand auf Basis einer PESM entstanden. Dieser ist mit Hilfe einer GUI in CANoe bedienbar. Diese bietet eine Drehzahl- und eine Drehmomentregelung mit einstellbaren Sollwerten, sowie eine vollautomatische Kennlinienerfassung. Außerdem wird eine Auswahl der verfügbaren Maschinendaten dargestellt.

Entgegen der Aufgabenstellung wurde keine Entwicklung in Matlab vorgenommen. Abgesehen von der graphischen Darstellung der aufgenommenen Kennlinien haben sich die Möglichkeiten von CANoe, das für den Betrieb des Teststandes ohnehin unerlässlich ist, mehr als ausreichend gezeigt. Die Einbindung einer weiteren Software eine hätte zu diesem Zeitpunkt unnötige Verkomplizierung bedeutet.

Außerdem konnte der Teststand ohne einen Prüfling noch nicht abschließend getestet werden. Dies wird jedoch in absehbarer Zeit geschehen.

4.2 Ausblick

In Zukunft wird dieser Teststand natürlich laufend an seine Aufgaben angepasst werden müssen. Dies betrifft zum Beispiel:

- eine flexible mechanische Verbindung zum Anschluss unterschiedlicher zu testender Maschinen
- weitere elektrische Anschlüsse zum Test vom unterschiedlichen Energiequellen
- ein verbessertes Not-Aus-System
- die Kühlmittelpumpe lässt sich auch durch eine PWM ansteuern, dies würde die Belastung der 12V Batterie und die Geräuschentwicklung verringern

Eine kleine Verbesserung der GUI bietet sich für die nahe Zukunft an: die Vorgabe der Sollwerte als Funktion würde eine bessere Betrachtung des Reglerverhaltens ermöglichen.

Literatur

[Berga] Johnson Controls Autobatterie GmbH: Gesamtkatalog, http://www.bvd24.de/wp-content/files_mf/1335436074Berga_Application.pdf, 2011/2012

[Bosch] Robert Bosch GmbH: Technische Kundenunterlage INV-CON.E-2.3 and SMG-180.1.3, April 2014

[CAPL] Vector Informatik GmbH: Steuergerätetests effizienter programmieren – Basics, Tipps und Tricks beim Einsatz von CAPL Teil 1: CAPL-Basics, https://assets.vector.com/cms/content/know-how/_technical-articles/CAPL_1_CANNewsletter_201406_PressArticle_DE.pdf, Ausgabe 2/2014

[dataTec] dataTec AG: Datenblatt | Elektro-Automatik PSI-9000 3U Programmierbare Hochleistungs-DC-Netzgeräte, https://www.datatec.de/media/pdf/f8/6a/1d/EA_PSI9000-3U.pdf, Dezember 2018

[Donath] Donath, Christian: Konzeption und Entwicklung der Kommunikationsstruktur des Elektrorennwagens des TMM, Hochschule Mittweida, University of Applied Sciences, Fakultät Elektro- und Informationstechnik, Praktikumsbericht 2018

[Nestler] Nestler, Marc: Realitätsnahe Ansteuerung der Antriebskomponenten eines Elektrorennfahrzeuges (FSE) auf dem Motorenprüfstand, als Vorstufe eines Fahrsimulators, Hochschule Mittweida, University of Applied Sciences, Fakultät Elektro- und Informationstechnik, Masterarbeit, 2015

[Thormann] Thormann, Christian: Entwicklung des Energie-Managements für ein Elektrofahrzeug (FSE) und Visualisierung der energetischen Zustandsgrößen zur Motorenprüfstandsüberwachung, Hochschule Mittweida, University of Applied Sciences, Fakultät Elektro- und Informationstechnik, Masterarbeit, 2015

[Vector] Vector Informatik GmbH: VN1600 Interface Familie Handbuch, https://assets.vector.com/cms/content/products/VN16xx/docs/VN1600_Interface_Family_Manual_DE.pdf, 2017

Anlagen

Anhang 1: Schaltplan

Anhang 2: CAPL Skript ECU_sim.can

Anhang 3: CAPL Skript BMS_sim.can

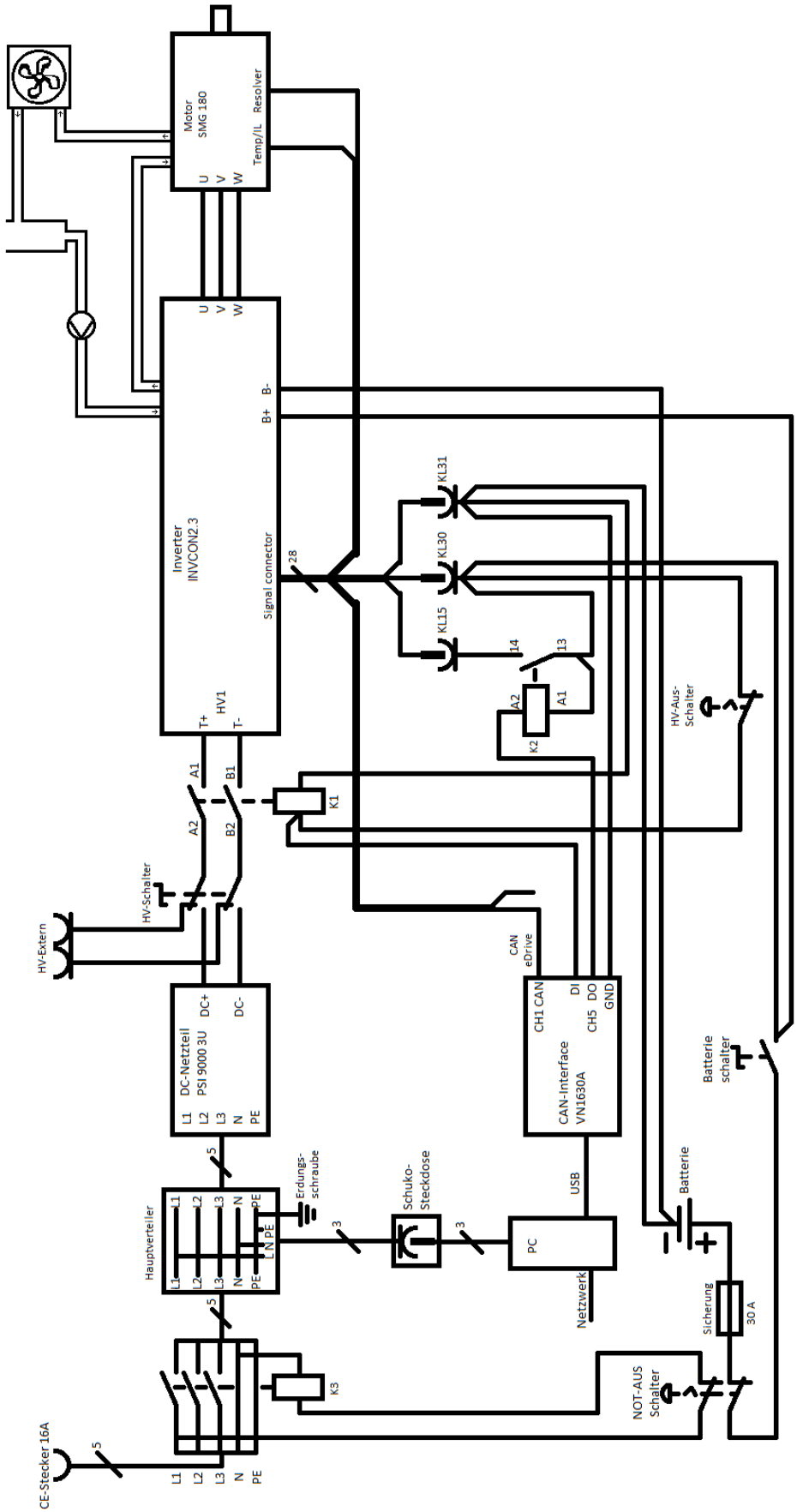
Anhang 4: CAPL Skript Init.cin

Anhang 5: CAPL Skript Functions.cin

Anhang 6: CAPL Skript Test.can

Anhang 7: Kurzanleitung Kennlinienaufnahme

Anhang 1: Schaltplan



Anhang 2: CAPL Skript ECU_sim.can

```
/*@!Encoding:1252*/
includes
{
  #include "Init.cin"
}

variables
{
  msTimer timer_10_1;
  dword filehandle = 0;
  char datensatz[64];
  char dateiname[34] = "Test.txt";
  long dt[9];             //Datum und Uhrzeit

  //Reglervariablen
  long e;                 //Regeldifferenz
  long eI;                //Zwischenspeicher Integrator
  long eD;                //Zwischenspeicher Differenzierer
  double uP;              //P-Anteil
  double uI;              //I-Anteil
  double uD;              //D-Anteil

  //Variablen Kennlinienaufnahme
  long rampenzeit;        //Anzahl der 10ms-Schritte
  long rampenstep;        //Schrittweite in 1/min pro 10ms-Schritt
  long nenndrehzahl;      //Zwischenspeicher für Ausgangspunkt
  int delta = 5;          //Toleranz für "Stillstand" in 1/min
}


//!!!!!!!!!!!!!!!!!!!!!!
// timer
//!!!!!!!!!!!!!!!!!!!!!!

on timer timer_10_1
{
  //Ansteuerung Inverter
  if (@Teststand::Aktivierung == 1)
  {
    if (@Teststand::Betriebsart == 0)                                   //Betriebsart Drehmomentregelung
    {
      VCU2INV_04.DesiredTorque.phys = @Teststand::Solldrehmoment;       //Drehmomentanforderung freigegeben
    }
    else                                                                //Betriebsart Drehzahlregelung
    {
      VCU2INV_04.DesiredTorque.phys = Drehzahlregler(@Teststand::Solldrehzahl); //Aufruf Drehzahlregler
    }
  }
  else
  {
    VCU2INV_04.DesiredTorque.phys = 0;                                  //Drehmomentanforderung gesperrt
    VCU2INV_04.DesiredSpeed.phys = 0;                                   //Drehzahlanforderung gesperrt
  }

  //Kennlinienaufnahme
  if (@Teststand::Kennlinie == 1)                                       //Wenn Kennlinienaufnahme aktiv
  {
    @Teststand::Solldrehzahl = @Teststand::Solldrehzahl - rampenstep;   //Verringere Solldrehzahl
    if (abs(@Teststand::Solldrehzahl) <= abs(rampenstep))               //Begrenze Solldrehzahl auf min 0 1/min
      @Teststand::Solldrehzahl = 0;
    if (abs($SpeedActual.phys) <= delta)                                //Bei Stillstand beende Kennlinienaufnahme
      @Teststand::Kennlinie = 0;
    if (filehandle != 0)
    {
      snprintf(datensatz, 64, "%.0f %.0f \n", $SpeedActual.phys, $OutputTorqueActual.phys);
      filePutString(datensatz, 64, filehandle);
    }
  }

  //Ansteuerung DC/DC-Wandler
  if (@Teststand::LV_laden == 1 && $VoltageHV_Actual.phys > 300)
  {
    VCU2DC_01.ModeRequest = VtSig_ModeRequest::Buck;                    //LV-Batterie laden
  }
  else
  {
    VCU2DC_01.ModeRequest = VtSig_ModeRequest::OFF;                     //LV-Batterie nicht laden
  }
  VCU2DC_01.VoltageDemandLV.phys = @Teststand::LV_Sollspannung;         //Spannungsvorgabe DC/DC-Wandler

  //Nachrichten Ausgabe
  VCU2DC_01.MessageCounter_130 = set_messageCounter(VCU2DC_01.id);
  VCU2DC_01.Checksum_130 = calc_Checksum_130();
  VCU2INV_04.MessageCounter_100 = set_messageCounter(VCU2INV_04.id);
  VCU2INV_04.Checksum_100 = calc_Checksum_100();

  output(VCU2DC_01);
  output(VCU2INV_04);
}


//!!!!!!!!!!!!!!!!!!!!!!
// on CAPL-function
//!!!!!!!!!!!!!!!!!!!!!!

on start
{
  init_VCU_DC_01();                                                     //Initialisierung der Nachrichten
  init_VCU_INV_04();

  timer_10_1.cancel();                                                  //Initialisierung des Timers
  setTimerCyclic(timer_10_1, 5, 10);
  setWritePath("D:\\Logging Dateien");                                  //Initialisierung des Dateipfades
  write("Initialisierung VCU durchgeführt");

  enableControl("Teststand_1", "Auswahl_Betriebsart", 0);               //Initialisierung der Panelelemente
  enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);
  enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
  enableControl("Teststand_1", "Button_Kennlinie", 0);
  write("Initialisierung Panel durchgeführt");
}
```

```
//!!!!!!!!!!!!!!!!!!!!!!!!
// on sysvar
//!!!!!!!!!!!!!!!!!!!!!!!!

on sysvar Teststand::Aktivierung
{
  if (@this == 1)
  {
    if (@IO::VN1600_1::DIN0 == 1 && $State == VtSig_State::Standby)
    {
      @IO::VN1600_1::DOUT = 1;                                          //KL15 einschalten
      VCU2INV_04.RequestedState = VtSig_RequestedState::TrqCtrl;        //Inverter aktivieren
      @Teststand::Betriebsart = 1;                                      //Betriebsart Drehzahlregelung
      @Teststand::Solldrehzahl = $SpeedActual.phys;
      enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);          //Panelelemente aktivieren
      enableControl("Teststand_1", "Auswahl_Betriebsart", 1);
      enableControl("Teststand_1", "Button_Kennlinie", 1);
    }
    else
    {
      @Teststand::Aktivierung = 0;
    }
  }
  else
  {
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;          //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                            //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                                  //kein Drehmoment mehr anfordern
    @Teststand::Kennlinie = 0;                                          //Kennlinienaufnahme beenden
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);          //Panelelemente deaktivieren
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
    enableControl("Teststand_1", "Auswahl_Betriebsart", 0);
    enableControl("Teststand_1", "Button_Kennlinie", 0);
  }
}

on sysvar IO::VN1600_1::DIN0                                            //Eingang HV-Aus
{
  if (@this == 0)
  {
    @Teststand::Aktivierung = 0;
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;          //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                            //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                                  //kein Drehmoment mehr anfordern
  }
}

on sysvar Teststand::Betriebsart
{
  if (@this == 0)                                                       //Betriebsart Drehmomentregelung
  {
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 1);
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);
    enableControl("Teststand_1", "Button_Kennlinie", 0);
    @Teststand::Solldrehmoment = $OutputTorqueActual.phys;
    @Teststand::Solldrehzahl = 0;
  }
  else                                                                  //Betriebsart Drehzahlregelung
  {
    enableControl("Teststand_1", "Eingabe_Solldrehmoment", 0);
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);
    enableControl("Teststand_1", "Button_Kennlinie", 1);
    @Teststand::Solldrehmoment = 0;
    @Teststand::Solldrehzahl = $SpeedActual.phys;
    eI = 0;                                                             //Zwischenspeicher Regler leeren
    eD = 0;
  }
}

on sysvar Teststand::Kennlinie
{
  if (@this == 0)                                                       //Ende oder Abbruch
  {
    if (@Teststand::Aktivierung == 1)
    {
      enableControl("Teststand_1", "Eingabe_Solldrehzahl", 1);          //Rückkehr zu Drehzahlregelung
      @Teststand::Solldrehzahl = nenndrehzahl;
    }
    enableControl("Teststand_1", "Eingabe_Rampe", 1);
    fileClose(filehandle);
  }
  else                                                                  //Beginn der Kennlinienaufnahme
  {
    enableControl("Teststand_1", "Eingabe_Solldrehzahl", 0);            //deaktivieren der Panelelemente
    enableControl("Teststand_1", "Eingabe_Rampe", 0);
    @Teststand::Solldrehzahl = $SpeedActual.phys;
    nenndrehzahl = $SpeedActual.phys;                                   //Zwischenspeichern der aktuellen Drehzahl
    rampenzeit = @Teststand::Rampe * 100;
    rampenstep = _floor(@Teststand::Solldrehzahl / rampenzeit);         //Berechnen der Schrittweite
    if ((rampenstep < 1) && (rampenstep > -1))                          //Schrittweite mindestens +-1
    {
      rampenstep = sign(@Teststand::Solldrehzahl);
      write("Rampe zu lang für diese Drehzahl! Neue Rampendauer %g s.", abs(@Teststand::Solldrehzahl / 100.));
    }
    else
    {
      write("Berechnete Rampendauer: %g s.", abs((@Teststand::Solldrehzahl / rampenstep) / 100.));
    }
    getLocalTime(dt);                                                   //Vorbereiten der Logging-Datei
    snprintf(dateiname, 34, "Kennlinie %d-%d-%d_%d-%d-%d.txt", (dt[5]+1900), dt[4], dt[3], dt[2], dt[1], dt[0]);
    filehandle = openFileWrite(dateiname, 2);
    if (filehandle != 0)
    {
      snprintf(datensatz, 64, "Drehzahl in 1/min\tDrehmoment in Nm\n");
      filePutString(datensatz, 64, filehandle);
    }
  }
}

//!!!!!!!!!!!!!!!!!!!!!!!!
// on message
//!!!!!!!!!!!!!!!!!!!!!!!!


on message INV_VCU_01
{
  if (this.State == VtSig_State::Failure)
  {
    @Teststand::Aktivierung = 0;
    VCU2INV_04.RequestedState = VtSig_RequestedState::Standby;          //Inverter deaktivieren
    @IO::VN1600_1::DOUT = 0;                                            //KL15 abschalten
    VCU2INV_04.DesiredTorque.phys = 0;                                  //kein Drehmoment mehr anfordern
  }
}


//!!!!!!!!!!!!!!!!!!!!!!!!
// functions
//!!!!!!!!!!!!!!!!!!!!!!!!

long Drehzahlregler (long Sollwert)
{
  e = Sollwert - $SpeedActual.phys;                                     //Regeldifferenz bilden
  eI = eI + e;                                                          //Integration
  uP = e * @Regler::kP;                                                 //P-Regler
  uI = eI * @Regler::kI;                                                //I-Regler
  uD = (e - eD) * @Regler::kD;                                          //D-Regler
  eD = e;                                                               //Subtrahend zwischenspeichern

  return (uP+uI+uD);                                                    //Regler kombinieren
}

long sign (long x)                                                      //Vorzeichenbestimmung
{
  if (x == 0) return 0;
  else return (x / abs(x));
}
```

Anhang 3: CAPL Skript BMS_sim.can

```
/*@!Encoding:1252*/
includes
{
  #include "Init.cin"
}

variables
{
  msTimer timer_10_2;
  msTimer timer_20_2;
  msTimer timer_100_2;
  msTimer timer_500_2;
}

on timer timer_10_2
{
  BMS2VCUINV_01.MessageCounter_6E = set_messageCounter(BMS2VCUINV_01.id);
  BMS2VCUINV_01.Checksum_6E = calc_Checksum_6E();
  VCU2INV_01.MessageCounter_2F0 = set_messageCounter(VCU2INV_01.id);
  VCU2INV_01.Checksum_2F0 = calc_Checksum_2F0();
  output(BMS2VCUINV_01);
  output(VCU2INV_01);
  output(VCU2INV_05);
  output(VCU2INV_06);
}

on timer timer_20_2
{
  PSM2BMSINV_01_GW.MessageCounter_A = set_messageCounter(PSM2BMSINV_01_GW.id);
  PSM2BMSINV_01_GW.Checksum_A = calc_Checksum_A();
  output(PSM2BMSINV_01_GW);
}

on timer timer_100_2
{
  BMS2VCUINV_02.MessageCounter_78 = set_messageCounter(BMS2VCUINV_02.id);
  BMS2VCUINV_04.MessageCounter_8C = set_messageCounter(BMS2VCUINV_04.id);
  BMS2VCUINV_07.MessageCounter_7D = set_messageCounter(BMS2VCUINV_07.id);
  BMS2VCUINV_07.Checksum_7D = calc_Checksum_7D();
  VCU2INV_03.MessageCounter_2F2 = set_messageCounter(VCU2INV_03.id);
  VCU2INV_03.Checksum_2F2 = calc_Checksum_2F2();
  output(BMS2VCUINV_02);
  output(BMS2VCUINV_04);
  output(BMS2VCUINV_07);
  output(VCU2INV_03);
  output(VCU2INV_02);
}

on timer timer_500_2
{
  BMS2VCUINV_05.MessageCounter_96 = set_messageCounter(BMS2VCUINV_05.id);
  BMS2VCUINV_05.Checksum_96 = calc_Checksum_96();
  output(BMS2VCUINV_05);
  output(BCM2INV_01_GW);
}

on start
{
  // Rest in ECU
  init_BCM_INV_01_GW();
  init_BMS_VCU_INV_01();
  init_BMS_VCU_INV_02();
  init_BMS_VCU_INV_04();
  init_BMS_VCU_INV_05();
  init_BMS_VCU_INV_07();
  init_PSM_BMS_INV_01_GW();
  init_VCU_INV_01();
  init_VCU_INV_02();
  init_VCU_INV_03();
  init_VCU_INV_05();
  init_VCU_INV_06();
  cancelTimer(timer_10_2);
  setTimerCyclic(timer_10_2, 10, 10);
  cancelTimer(timer_20_2);
  setTimerCyclic(timer_20_2, 20, 20);
  cancelTimer(timer_100_2);
  setTimerCyclic(timer_100_2, 100, 100);
  cancelTimer(timer_500_2);
  setTimerCyclic(timer_500_2, 500, 500);
  write("Initialisierung BMS durchgeführt");
}
```

Anhang 4: CAPL Skript Init.cin

```
/*@!Encoding:1252*/
// initialisation of all rx-messages of the inverter

includes
{
  #include "Functions.cin"
}

variables
{
  //Rx-Messages
  message BMS_VCU_INV_01 BMS2VCUINV_01;
  message BMS_VCU_INV_02 BMS2VCUINV_02;
  message BMS_VCU_INV_04 BMS2VCUINV_04;
  message BMS_VCU_INV_05 BMS2VCUINV_05;
  message BMS_VCU_INV_07 BMS2VCUINV_07;
  message BCM_INV_01_GW BCM2INV_01_GW;
  message PSM_BMS_INV_01_GW PSM2BMSINV_01_GW;
  message VCU_DC_01 VCU2DC_01;
  message VCU_INV_01 VCU2INV_01;
  message VCU_INV_02 VCU2INV_02;
  message VCU_INV_03 VCU2INV_03;
  message VCU_INV_04 VCU2INV_04;
  message VCU_INV_05 VCU2INV_05;
  message VCU_INV_06 VCU2INV_06;

  //Tx-Messages
  message DC_VCU_01 DC2VCU_01;
  message DC_VCU_02 DC2VCU_02;
  message DC_VCU_03 DC2VCU_03;
  message INV_VCU_01 INV2VCU_01;
  message INV_VCU_02 INV2VCU_02;
  message INV_VCU_03 INV2VCU_03;
  message INV_VCU_04 INV2VCU_04;
  message INV_VCU_05 INV2VCU_05;

  word UDC_MAX = 410;
  word UDC_MIN = 300;
  word IDC_MAX = 30;
  word IDC_MIN = 0;
}

void init_BCM_INV_01_GW(void)
{
  // source: VCU | sink: Inverter
  BCM2INV_01_GW.ReleaseTankClapDemand_GW = 0x02;     // senseless without tank
  BCM2INV_01_GW.UnplugDemand_GW = 0x00;              // senseless without tank

  output(BCM2INV_01_GW);
}

void init_BMS_VCU_INV_01(void)
{
  // source: BMS | sink: Inverter, VCU, REXECU
  BMS2VCUINV_01.MessageCounter_6E = 0x00;
  BMS2VCUINV_01.ActualCurrent = vtSig_ActualCurrent::Init;
  BMS2VCUINV_01.ActualVoltage = vtSig_ActualVoltage::Init;
  BMS2VCUINV_01.ActualState = vtSig_ActualState::Initializing;     // wenn nicht geht eventuell in 0x01=HVActive
  BMS2VCUINV_01.IsoMeasurementActive = vtSig_IsoMeasurementActive::Init;
  BMS2VCUINV_01.ErrorIsolation = 0;
  BMS2VCUINV_01.ErrorEmergencyOffCrash = 0;
  BMS2VCUINV_01.ErrorEmergencyOffPilot = 0;
  BMS2VCUINV_01.ErrorHvBattEmergencyOffReq = 0;
  BMS2VCUINV_01.ErrorHvBattEmergencyOff = 0;
  BMS2VCUINV_01.ErrorRelayWelded = 0;
  BMS2VCUINV_01.ErrorLimitedPower = 0;
  BMS2VCUINV_01.ErrorRelayOpen = 0;
  BMS2VCUINV_01.ContactorState = vtSig_ContactorState::HVBattCntctr_Init;

  BMS2VCUINV_01.Checksum_6E = calc_Checksum_6E();
  output(BMS2VCUINV_01);
}

void init_BMS_VCU_INV_02(void)
{
  // source: BMS | sink: Inverter, VCU
  BMS2VCUINV_02.MessageCounter_78 = 0x00;
  BMS2VCUINV_02.MaxChargingCurrentContinuous.phys = 1022;
  BMS2VCUINV_02.AvailableMaxDisChgCurrContinuous.phys = 1022;
  BMS2VCUINV_02.MaxChargingCurrentLongTerm.phys = 1022;
  BMS2VCUINV_02.AvailableMaxDisChgCurrLongTerm.phys = 1022;
  BMS2VCUINV_02.MaxChargingCurrentShortTerm.phys = 1022;
  BMS2VCUINV_02.AvailableMaxDisChgCurrShortTerm.phys = 1022;

  output(BMS2VCUINV_02);
}

void init_BMS_VCU_INV_04(void)
{
  // source: BMS | sink: Inverter, VCU (ActualTemperatureCellTemperature nur an VCU)
  BMS2VCUINV_04.MessageCounter_8C = 0x00;
  BMS2VCUINV_04.MaxChargingPowerContinuous.phys = 0;             // not used by the inverter, if it doesnt work: .phys=306.9 for all signals
  BMS2VCUINV_04.AvailableMaxDisChgPwrContinuous.phys = 0;        // not used by the inverter
  BMS2VCUINV_04.MaxChargingPowerLongTerm.phys = 0;               // not used by the inverter
  BMS2VCUINV_04.AvailableMaxDisChgPowerLongTerm.phys = 0;        // not used by the inverter
  BMS2VCUINV_04.MaxChargingPowerShortTerm.phys = 0;              // not used by the inverter
  BMS2VCUINV_04.AvailableMaxDisChgPowerShortTerm.phys = 0;       // not used by the inverter

  output(BMS2VCUINV_04);
}

```

```
void init_BMS_VCU_INV_05(void)
{
  // source: BMS | sink: Inverter, VCU, REXECU
  BMS2VCUINV_05.MessageCounter_96 = 0x00;
  BMS2VCUINV_05.MaxChargingVoltage.phys = UDC_MAX;
  BMS2VCUINV_05.AvailableMinDischargingVoltage.phys = UDC_MIN;
  BMS2VCUINV_05.ActualTemperatureCellTemperature.phys = 0;

  BMS2VCUINV_05.Checksum_96 = calc_Checksum_96();

  output(BMS2VCUINV_05);
}

void init_BMS_VCU_INV_07(void)
{
  // source: BMS | sink: Inverter, VCU, REXECU
  BMS2VCUINV_07.MessageCounter_7D = 0x00;
  BMS2VCUINV_07.DesiredChargingMode = vtSig_DesiredChargingMode::Init;
  BMS2VCUINV_07.CurrentControlledDesiredCurrent.phys = 0;
  BMS2VCUINV_07.MaxChargingCurrent.phys = 255.9375;
  BMS2VCUINV_07.VoltageControlledDesiredVoltage.phys = vtSig_VoltageControlledDesiredVoltage::Init;
  BMS2VCUINV_07.RemainingFastChargingCycles.phys = vtSig_RemainingFastChargingCycles::Init;

  BMS2VCUINV_07.Checksum_7D = calc_Checksum_7D();

  output(BMS2VCUINV_07);
}

void init_PSM_BMS_INV_01_GW(void)
{
  // source: VCU | sink: Inverter, BMS
  PSM2BMSINV_01_GW.MessageCounter_A = 0x00;
  PSM2BMSINV_01_GW.CrashState_GW = 0;
  PSM2BMSINV_01_GW.CrashActuatorTest_GW = 0;

  PSM2BMSINV_01_GW.Checksum_A = calc_Checksum_A();

  output(PSM2BMSINV_01_GW);
}

void init_VCU_DC_01(void)
{
  // source: VCU | sink: Inverter
  VCU2DC_01.MessageCounter_130 = 0x00;
  VCU2DC_01.VoltageDemandHV.phys = 0;
  VCU2DC_01.CurrentDemandHV.phys = 0;
  VCU2DC_01.CurrentDemandLV.phys = 0;
  VCU2DC_01.VoltageDemandLV.phys = 14;
  VCU2DC_01.ModeRequest = vtSig_ModeRequest::OFF;

  VCU2DC_01.Checksum_130 = calc_Checksum_130();

  output(VCU2DC_01);
}

void init_VCU_INV_01(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_01.MessageCounter_2F0 = 0;
  VCU2INV_01.CurrentActualHVAux = 1;
  VCU2INV_01.IdcMax.phys = IDC_MAX;
  VCU2INV_01.IdcMin.phys = IDC_MIN;
  VCU2INV_01.UdcMax.phys = UDC_MAX;
  VCU2INV_01.UdcMin.phys = UDC_MIN;

  VCU2INV_01.Checksum_2F0 = calc_Checksum_2F0();

  output(VCU2INV_01);
}

void init_VCU_INV_02(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_02.FactorPContributionLarge.phys = 0;
  VCU2INV_02.FactorPContributionSmall.phys = 0;
  VCU2INV_02.MaxDeltaSpeed.phys = -16383;
  VCU2INV_02.MinTorqueEngineSpeed.phys = 220;
  VCU2INV_02.MaxTorqueEngineSpeed.phys = -220;

  output(VCU2INV_02);
}

void init_VCU_INV_03(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_03.MessageCounter_2F2 = 0;
  VCU2INV_03.MinTorqueLimit.phys = -220;
  VCU2INV_03.MaxTorqueGradient.phys = 250496;
  VCU2INV_03.MaxTorqueLimit.phys = 220;

  VCU2INV_03.Checksum_2F2 = calc_Checksum_2F2();

  output(VCU2INV_03);
}

void init_VCU_INV_04(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_04.MessageCounter_100 = 0;
  VCU2INV_04.DesiredSpeed.phys = 0;       // speedcontrol not implemented
  VCU2INV_04.RequestedState = vtSig_RequestedState::Init;
  VCU2INV_04.UdcSetP = vtSig_UdcSetP::Init;
  VCU2INV_04.DesiredTorque.phys = 0;
  VCU2INV_04.StHvbMaiRly = vtSig_StHvbMaiRly::Init;

  VCU2INV_04.Checksum_100 = calc_Checksum_100();

  output(VCU2INV_04);
}

void init_VCU_INV_05(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_05.FrqStrSetP.phys = -3840;
  VCU2INV_05.IsSetP.phys = 0;
  VCU2INV_05.agOffs.phys = 0;
  VCU2INV_05.bSetOfsAl = 0;
  VCU2INV_05.bUseFrqStrSynt = 0;
  VCU2INV_05.flgInvrCanActvnCmd = 1;      // Wakeup of the Inverter over CAN -> use for energysaving

  output(VCU2INV_05);
}

void init_VCU_INV_06(void)
{
  // source: VCU | sink: Inverter
  VCU2INV_06.SurgeDamperState = 0x00;     // 30.10.2013: not used by the inverter (now)
  // folgende Werte lt. Messung René Socher(Bosch) -> 0x1023=Error -> wenn nicht funktioniert, 0x1022 probieren
  VCU2INV_06.MaxTorqueDampgCtl = 1023;    // =Error
  VCU2INV_06.Treshold = 1023;             // =Error
  VCU2INV_06.SurgeDamperFactor.phys = 1.9688;

  output(VCU2INV_06);
}
```

Anhang 5: CAPL Skript Functions.cin

```
/*@!Encoding:1252*/
variables
{

}

byte calc_Checksum_6E()
{
  byte check_6E = 0x00;

  check_6E = BMS2VCUINV_01.byte(1)^BMS2VCUINV_01.byte(2)^BMS2VCUINV_01.byte(3)^BMS2VCUINV_01.byte(4)
    ^BMS2VCUINV_01.byte(5)^BMS2VCUINV_01.byte(6);

  return check_6E;
}

byte calc_Checksum_96()
{
  byte check_96 = 0x00;

  check_96 = BMS2VCUINV_05.byte(1)^BMS2VCUINV_05.byte(2)^BMS2VCUINV_05.byte(3)^BMS2VCUINV_05.byte(4);

  return check_96;
}

byte calc_Checksum_7D()
{
  byte check_7D = 0x00;

  check_7D = BMS2VCUINV_07.byte(1)^BMS2VCUINV_07.byte(2)^BMS2VCUINV_07.byte(3)^BMS2VCUINV_07.byte(4)
    ^BMS2VCUINV_07.byte(5)^BMS2VCUINV_07.byte(6)^BMS2VCUINV_07.byte(7);

  return check_7D;
}

byte calc_Checksum_A()
{
  byte check_A = 0x00;

  check_A = PSM2BMSINV_01_GW.byte(1)^PSM2BMSINV_01_GW.byte(2)^PSM2BMSINV_01_GW.byte(3)^PSM2BMSINV_01_GW.byte(4)
    ^PSM2BMSINV_01_GW.byte(5)^PSM2BMSINV_01_GW.byte(6)^PSM2BMSINV_01_GW.byte(7);

  return check_A;
}

// DLC = 8, 7 bytes sind aber nur belegt -> relevant für Checksumme
byte calc_Checksum_130()
{
  byte check_130;// = 0x00;

  check_130 = VCU2DC_01.byte(1)^VCU2DC_01.byte(2)^VCU2DC_01.byte(3)^VCU2DC_01.byte(4)
    ^VCU2DC_01.byte(5)^VCU2DC_01.byte(6)^VCU2DC_01.byte(7);

  return check_130;
}

byte calc_Checksum_2F0()
{
  byte check_2F0 = 0x00;

  check_2F0 = VCU2INV_01.byte(1)^VCU2INV_01.byte(2)^VCU2INV_01.byte(3)^VCU2INV_01.byte(4)
    ^VCU2INV_01.byte(5)^VCU2INV_01.byte(6)^VCU2INV_01.byte(7);

  return check_2F0;
}

byte calc_Checksum_2F2()
{
  byte check_2F2 = 0x00;

  check_2F2 = VCU2INV_03.byte(1)^VCU2INV_03.byte(2)^VCU2INV_03.byte(3)^VCU2INV_03.byte(4)
    ^VCU2INV_03.byte(5)^VCU2INV_03.byte(6)^VCU2INV_03.byte(7);

  return check_2F2;
}

byte calc_Checksum_100()
{
  byte check_100;

  check_100 = VCU2INV_04.byte(1)^VCU2INV_04.byte(2)^VCU2INV_04.byte(3)^VCU2INV_04.byte(4)
    ^VCU2INV_04.byte(5)^VCU2INV_04.byte(6)^VCU2INV_04.byte(7);

  return check_100;
}
```

```
byte set_messageCounter(dword msg_id)
{
  switch(msg_id)
  {
    case 0x6E:
      return MsgCount(BMS2VCUINV_01.MessageCounter_6E);
      break;
    case 0x78:
      return MsgCount(BMS2VCUINV_02.MessageCounter_78);
      break;
    case 0x8C:
      return MsgCount(BMS2VCUINV_04.MessageCounter_8C);
      break;
    case 0x96:
      return MsgCount(BMS2VCUINV_05.MessageCounter_96);
      break;
    case 0x7D:
      return MsgCount(BMS2VCUINV_07.MessageCounter_7D);
      break;
    case 0xA:
      return MsgCount(PSM2BMSINV_01_GW.MessageCounter_A);
      break;
    case 0x130:
      return MsgCount(VCU2DC_01.MessageCounter_130);
      break;
    case 0x2F0:
      return MsgCount(VCU2INV_01.MessageCounter_2F0);
      break;
    case 0x2F2:
      return MsgCount(VCU2INV_03.MessageCounter_2F2);
      break;
    case 0x100:
      return MsgCount(VCU2INV_04.MessageCounter_100);
      break;
  }
  write("set_messageCounter(): Botschaft unbekannt");
  return 0;
}

char MsgCount(char count_val)
{
  if (count_val < 15)
  {
    count_val++;
    return count_val;
  }
  else
  {
    return 0;
  }
}
```

Anhang 6: CAPL Skript Test.can

```
/*@!Encoding:1252*/
includes
{

}

variables
{
//  message BMS_VCU_INV_01 BMS2VCUINV_01;
//  message BMS_VCU_INV_02 BMS2VCUINV_02;
//  message BMS_VCU_INV_04 BMS2VCUINV_04;
//  message BMS_VCU_INV_05 BMS2VCUINV_05;
//  message BMS_VCU_INV_07 BMS2VCUINV_07;
//  message BCM_INV_01_GW BCM2INV_01_GW;
//  message PSM_BMS_INV_01_GW PSM2BMSINV_01_GW;
//  message VCU_DC_01 VCU2DC_01;
//  message VCU_INV_01 VCU2INV_01;
//  message VCU_INV_02 VCU2INV_02;
//  message VCU_INV_03 VCU2INV_03;
//  message VCU_INV_04 VCU2INV_04;
//  message VCU_INV_05 VCU2INV_05;
//  message VCU_INV_06 VCU2INV_06;

  byte MC_BMS_VCU_INV_01 = 0x0F;
  byte MC_BMS_VCU_INV_02 = 0x0F;
  byte MC_BMS_VCU_INV_04 = 0x0F;
  byte MC_BMS_VCU_INV_05 = 0x0F;
  byte MC_BMS_VCU_INV_07 = 0x0F;
  byte MC_PSM_BMS_INV_01_GW = 0x0F;
  byte MC_VCU_DC_01 = 0x0F;
  byte MC_VCU_INV_01 = 0x0F;
  byte MC_VCU_INV_03 = 0x0F;
  byte MC_VCU_INV_04 = 0x0F;
}

on start
{
  write("Initialisierung Test durchgeführt");
}

on message BMS_VCU_INV_01
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6);
  if (temp != 0)
  {
    write("Checksumme ID=6E (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_6E;
  temp = MC_Neu - MC_BMS_VCU_INV_01;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=6E (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_01 = MC_Neu;
}

on message BMS_VCU_INV_02
{
  byte temp;
  byte MC_Neu;

  MC_Neu = this.MessageCounter_78;
  temp = MC_Neu - MC_BMS_VCU_INV_02;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=78 (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_02 = MC_Neu;
}

on message BMS_VCU_INV_04
{
  byte temp;
  byte MC_Neu;

  MC_Neu = this.MessageCounter_8C;
  temp = MC_Neu - MC_BMS_VCU_INV_04;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=8C (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_04 = MC_Neu;
}

on message BMS_VCU_INV_05
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4);

  if (temp != 0)
  {
    write("Checksumme ID=96 (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_96;
  temp = MC_Neu - MC_BMS_VCU_INV_05;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=96 (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_05 = MC_Neu;
}

on message BMS_VCU_INV_07
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=7D (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_7D;
  temp = MC_Neu - MC_BMS_VCU_INV_07;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=7D (0x%X) fehlerhaft!", temp);
  }
  MC_BMS_VCU_INV_07 = MC_Neu;
}

```

```
on message PSM_BMS_INV_01_GW
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=A (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_A;
  temp = MC_Neu - MC_PSM_BMS_INV_01_GW;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=A (0x%X) fehlerhaft!", temp);
  }
  MC_PSM_BMS_INV_01_GW = MC_Neu;
}

on message VCU_DC_01
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=130 (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_130;
  temp = MC_Neu - MC_VCU_DC_01;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=130 (0x%X) fehlerhaft!", temp);
  }
  MC_VCU_DC_01 = MC_Neu;
}

on message VCU_INV_01
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=2F0 (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_2F0;
  temp = MC_Neu - MC_VCU_INV_01;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=2F0 (0x%X) fehlerhaft!", temp);
  }
  MC_VCU_INV_01 = MC_Neu;
}

on message VCU_INV_03
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=2F2 (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_2F2;
  temp = MC_Neu - MC_VCU_INV_03;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=2F2 (0x%X) fehlerhaft!", temp);
  }
  MC_VCU_INV_03 = MC_Neu;
}

on message VCU_INV_04
{
  byte temp;
  byte MC_Neu;

  temp = this.byte(0)^this.byte(1)^this.byte(2)^this.byte(3)^this.byte(4)^this.byte(5)^this.byte(6)^this.byte(7);

  if (temp != 0)
  {
    write("Checksumme ID=100 (0x%X) fehlerhaft!", temp);
  }

  MC_Neu = this.MessageCounter_100;
  temp = MC_Neu - MC_VCU_INV_04;
  if ((temp != 0xF1) && (temp != 0x01))
  {
    write("Message Counter ID=100 (0x%X) fehlerhaft!", temp);
  }
  MC_VCU_INV_04 = MC_Neu;
}
```

Anhang 7: Kurzanleitung Kennlinienaufnahme

Schritt 1:

Versichere dich, dass der NOT-AUS- und der HV-Aus-Knopf unbetätigt sind, der HV-Umschalter auf „Intern“ steht und das Netzgerät und der Batterieschalter abgeschalten sind.

Schritt 2:

Verbinde den Teststand mit dem roten CE-Stecker mit dem Energienetz.

Schritt 3:

Starte den Rechner. Öffne CANoe und das Panel „Teststand_1“.

Schritt 4:

Schalte mit dem Batterieschalter die Steuerspannung ein.
Achtung! Schalte die Steuerspannung nicht vor dem vorgesehenen Zeitpunkt wieder ab! Das Abschalten der Steuerspannung während noch HV-Spannung anliegt kann zur Zerstörung des Inverters führen!

Schritt 5:

Schalte das DC-Netzgerät ein. Stelle eine Ausgangsspannung von 400 V ein. Schalte den Ausgang auf „On“.

Schritt 6:

Verwende den Button „START“ im CANoe oder dem Panel, um eine Messung zu beginnen.

Schritt 7:

Stelle zu diesem Testaufbau passende Regler-Parameter ein.

Schritt 8:

Starte die zu testende Maschine und stelle sie auf Leerlaufdrehzahl ein.

Schritt 9:

Aktiviere den Teststand mit dem Button „Aktivierung“.

Schritt 10:

Variiere die Solldrehzahl eventuell so, dass das angezeigte Drehmoment möglichst 0 ist.

Schritt 11:

Starte die Aufnahme der Kennlinie mit dem Button „Kennlinie aufnehmen“.

Schritt 12:

Warte bis die Maschinen sich wieder auf Leerlaufdrehzahl eingestellt haben.

Schritt 13:

Im Ordner D:\Logging Dateien befindet sich nun eine Datei mit dem Namen „Kennlinie [Datum]_[Uhrzeit].txt“. Diese enthält die gemessenen Punkte der Kennlinie in Tabellenform.

Schritt 14:

Bei Bedarf: Verändere die Regler-Parameter und/oder die Länge der Rampe und wiederhole die Schritte 11 – 13.

Schritt 15:

Deaktiviere den Teststand mit dem Button „Aktivierung“.

Schritt 16:

Schalte die zu testende Maschine ab.

Schritt17:

Betätige den HV-Aus-Taster.

Schritt 18:

Deaktiviere das DC-Netzteil: Ausgang auf „Off“ und abschalten.

Schritt 19:

Beende die Messung mit dem Button „STOP“, schließe CANoe und fahre den Rechner herunter.

Schritt 20:

Schalte mit dem Batterieschalter die Steuerspannung ab.

Schritt 21:

Trenne den Teststand vom Netz.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 31.08.2019

Christian Donath